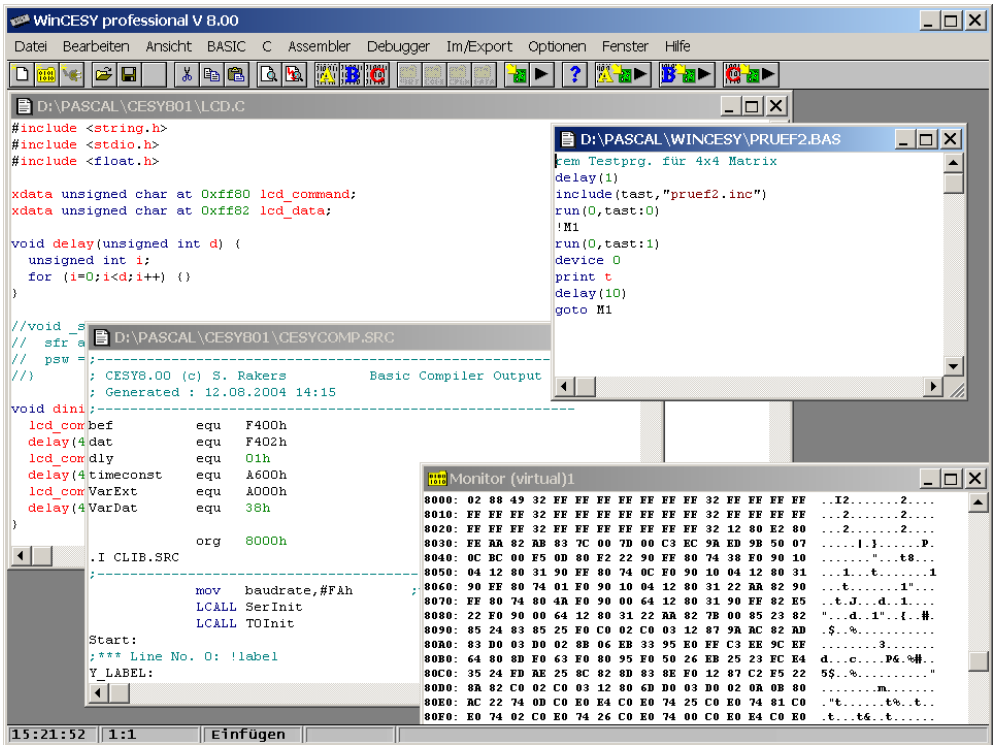
Dr. Sven Rakers

CESY 8

Das Cross-Entwicklungssystem für alle Mikrocontroller der 80x51-Familie


WinCESY professional V 8.00
Datei Bearbeiten Ansicht BASIC C Assembler Debugger Im/Export Optionen Fenster Hilfe
D:\PASCAL\CESY801\LCD.C
#include <string.h>
#include <stdio.h>
#include <float.h>
xdata unsigned char at 0xff80 lcd_command;
xdata unsigned char at 0xff82 lcd_data;
void delay(unsigned int d) {
unsigned int i;
for (i=0;i<d;i++) {}
}
D:\PASCAL\WINCESY\PRUEF2.BAS
rem Testprg. für 4x4 Matrix
delay(1)
include(tast,"pruef2.inc")
run(0,tast:0)
!M1
run(0,tast:1)
device 0
print t
delay(10)
goto M1
D:\PASCAL\CESY801\CESYCOMP.SRC
; CESY8.00 (c) S. Rakers Basic Compiler Output
; Generated : 12.08.2004 14:15
bef equ F400h
dat equ F402h
dly equ 01h
timeconst equ A600h
VarExt equ A000h
VarDat equ 38h
org 8000h
.I CLIB.SRC
mov baudrate,#FAh
LCALL SerInit
LCALL T0Init
Start:
;*** Line No. 0: !label
Y_LABEL:
Monitor (virtual)1
8000: 02 88 49 32 FF FF FF FF FF FF 32 FF FF FF FF ..I2.....2....
8010: FF FF FF 32 FF FF FF FF FF FF 32 FF FF FF FF ...2.....2....
8020: FF FF FF 32 FF FF FF FF FF FF 32 12 80 E2 80 ...2.....2....
8030: FE AA 82 AB 83 7C 00 7D 00 C3 EC 9A ED 9B 50 07 .....|.}......P.
8040: 0C BC 00 F5 0D 80 F2 22 90 FF 80 74 38 F0 90 10 ....".t8...
8050: 04 12 80 31 90 FF 80 74 0C F0 90 10 04 12 80 31 ...1...t.......1
8060: 90 FF 80 74 01 F0 90 10 04 12 80 31 22 AA 82 90 ...t.......1"...
8070: FF 80 74 80 4A F0 90 00 64 12 80 31 90 FF 82 E5 ..t.J...d..1....
8080: 22 F0 90 00 64 12 80 31 22 AA 82 7B 00 85 23 82 "...d..1"..{..#.
8090: 85 24 83 85 25 F0 C0 02 C0 03 12 87 9A AC 82 AD .$..%...........
80A0: 83 D0 03 D0 02 8B 06 EB 33 95 E0 FF C3 EE 9C EF ....3.......
80B0: 64 80 8D F0 63 F0 80 95 F0 50 26 EB 25 23 FC E4 d...c....P&.%#..
80C0: 35 24 FD AE 25 8C 82 8D 83 8E F0 12 87 C2 F5 22 5$..%..........."
80D0: 8A 82 C0 02 C0 03 12 80 6D D0 03 D0 02 0A 0B 80 ....m.......
80E0: AC 22 74 0D C0 E0 E4 C0 E0 74 25 C0 E0 74 81 C0 ."t......t%..t..
80F0: E0 74 02 C0 E0 74 26 C0 E0 74 00 C0 E0 E4 C0 E0 .t...t&..t......
15:21:52 1:1 Einfügen

Benutzerhandbuch

- Stand: V8.0, September 2004 -

Inhalt

1 ÜBERBLICK 8

2 INSTALLATION UND ERSTE SCHRITTE 10

2.1 Installation 10
2.2 Installation des C-Compilers 11
2.3 Deinstallation 12
2.4 Starten von CESY 13
2.5 Kennenlernen von CESY 13
2.6 Probleme erkennen und lösen 21
2.7 Programme in ein EPROM brennen 22

3 CESY-REFERENZ 24

3.1 Die Benutzeroberfläche 24
3.2 Das Speicherkonzept 25
3.3 Vom Quelltext zum lauffähigen Programm 26
3.4 Das Datei-Menü 26
3.5 Der Editor 27
3.6 Der Assembler 29
3.7 Assembler-Syntax 30
3.8 Der Reassembler 39
3.9 Der Monitor 41
3.10 Der Debugger 42
3.11 Kommunikation mit dem Zielsystem 46
3.12 Das Terminal-Fenster 49

4 HARDWARE UND CESY-FIRMWARE 50

4.1 Hardware 50
4.2 Die CESY-Firmware 51

5 BASIC-COMPILER 54

5.1 Übersetzen 54
5.2 Einstellungen 54
5.3 Aufbau eines BASIC-Programms 56
5.4 BASIC-Referenz 58
5.5 Interrupts in BASIC 76
5.6 Besonderheiten bei C517/C509 Prozessoren 76
5.7 Fehlermeldungen 77

6 DER C-COMPILER 80

6.1 Stand der Entwicklung 80
6.2 Open Source, Urheberrecht und Garantieverzicht 80
6.3 Speicherbereiche 81

6.4 Compilieren 82
6.5 Projekt-Manager 83
6.6 Die Programmiersprache C 85
6.7 Spracherweiterungen und Besonderheiten 85
6.8 Weitere Dokumentation 90

7 DIVERSES 92

7.1 Eigenschaften 92
7.2 Blitzstart 92

8 NEUE SDCC-VERSION 93

9 BUG GEFUNDEN? 93

10 DER BEFEHLSSATZ DES 80X51 94

Das CESY-Handbuch

Dieses Handbuch soll Ihnen bei der Arbeit mit CESY als Referenz zum schnellen Nachschlagen dienen. Es enthält eine Auswahl der wichtigsten Informationen der Online-Hilfe.

Handbücher werden gewöhnlich in größeren Stückzahlen gedruckt. Das bedeutet allerdings, dass ein solches Handbuch nicht lange aktuell bleibt. CESY wird des öfteren überarbeitet und in einer neuen Versionsnummer angeboten. Jedes Mal neue Handbücher zu drucken wäre unrentabel und dazu umweltbelastend.

Aus diesen Gründen liefert diese Broschüre Standardinformationen, die mit dem Stand dieses Buches aktuell sind. Weitere Informationen finden Sie in der ständig aktualisierten Online-Hilfe.

Wenn Sie ein neues Update erhalten, sehen Sie bitte zunächst die README-Datei auf Ihrer Programmdiskette/-CD an. Dort finden Sie Hinweise auf neue Befehle, Änderungen, Bugfixes usw. Ausführlichere Informationen dazu sind dann in der Online-Hilfe enthalten.

Über Anregungen von der "Benutzerfront" sind wir jederzeit dankbar und werden sie, soweit für einen größeren Kreis nützlich, in das nächsterreichbare Update einbringen.

Das Handbuch ist in fünf Teile unterteilt:

- Installation und erste Schritte
- Referenzhandbuch
- Hardware & Firmware
- BASIC-Referenz
- C-Handbuch

Wir wünschen Ihnen viel Erfolg und eine angenehme Arbeit mit CESY!

1 Überblick

CESY ist eine komfortable Entwicklungsplattform, um Programme für Mikrocontroller der 80x51-Familie zu entwickeln. Dazu stehen ein leistungsfähiger Makroassembler und ein Basic-Compiler zur Verfügung. Der besondere Clou ist, dass CESY alle entstehenden Object-Codes nicht in irgendwelche Dateien auf der Festplatte packt, sondern einen 64 kB großen, sogenannten virtuellen Speicher emuliert, der die entstehenden Programmstücke aufnimmt. Den virtuellen Speicher können Sie mit dem Monitor nach Herzenslust einsehen und editieren.

Das Zielsystem (das Mikrocontroller-System, für welches Ihr Programm geschrieben wird) ist dabei eng mit der Plattform verbunden. Download (Übertragen von Programmcode vom Entwicklungssystem CESY an das Zielsystem) und Upload (Übertragen von Programmcode vom Zielsystem zum Entwicklungssystem CESY) sowie das Starten von Programmen erfolgen komfortabel und flexibel über Menüs. Den Speicher des Zielsystems haben Sie mit dem Monitor voll unter Kontrolle.

Assembler-Programme können Sie bequem mit dem Debugger auf Fehlerfreiheit testen. Dabei wird die entsprechende Zeile im Editor hervorgehoben, und Daten- sowie SFR-Register kontinuierlich angezeigt.

Wer seine Programme lieber in einer Hochsprache erstellt, hat mit dem BASIC-Compiler ein mächtiges Werkzeug zur Hand. Innerhalb von Minuten ist ein einfaches Programm erstellt, kompiliert, heruntergeladen und gestartet. Das BASIC ist durch eine neuentwickelte 24-bit Integer-Arithmetik extrem schnell. Eine einfache, an C angelehnte String-Verarbeitung vereinfacht eine eventuelle Benutzerführung, z.B. über die serielle Schnittstelle oder über ein ebenfalls unterstütztes LCD-Display. Das Konzept des BASIC lehnt sich stark an die Maschine an, z.B. durch fest an Speicherstellen gebundene Variablen. Dadurch wird der Austausch von Daten zwischen BASIC und eingebundenen Assemblerroutinen wesentlich erleichtert. Unser Tip: schreiben Sie den Programmrahmen, die Benutzerführung und arithmetik-intensive Routinen in BASIC und den Rest in Assembler. Sie werden staunen, welch professionelle Programme das Ergebnis sein werden.

Damit ein reibungsloses Zusammenspiel von CESY und unserem Flaggschiff „LAB-537“ möglich ist, wurden spezielle BASIC-Kommandos für ADC und das FLASH-EPROM in CESY aufgenommen.

Als Neuheit in Version 8 ist der beliebte Freeware-C-Compiler SDCC in CESY aufgenommen worden. Der Compiler ist voll in CESY integriert, so dass der Benutzer, wie in Basic oder Assembler gewohnt, seine C-Programme und –Projekte nach dem Übersetzen sofort im virtuellen Speicher zur Verfügung hat. Außerdem gibt es einen Projektmanager, mit dem komplexe Programme aus mehreren Dateien effizient verwaltet werden können.

a

2 Installation und erste Schritte

2.1 Installation

Zur Installation müssen die CESY-Programmdateien, die Beispiele und Quelltexte sowie eine DLL-Datei auf Ihre Festplatte kopiert werden. Ferner sind im Programm-Manager (Windows 3.1), bzw. im Startmenü (ab Windows 95) Einträge auf das CE-SY-Programm und die Hilfedatei zu setzen. Schließlich muss noch der externe C-Compiler SDCC installiert werden, falls auch in C programmiert werden soll.

Dank des Installationsprogramms geschehen die ersten beiden Schritte vollautoma-tisch. Sie müssen nur die gewünschten Verzeichnisse angeben, den Rest übernimmt das Installationsprogramm.

Legen Sie nun bitte Ihre CESY-Installationsdiskette oder -CD ein und starten Sie Ihr Windows-Betriebssystem.

2.1.1 Windows 3.1, NT 3:

Wählen Sie im Programmanager den Menüpunkt "Ausführen" und starten Sie das Installationsprogramm "SETUP.EXE" auf der Installationsdiskette, bzw. CD.

2.1.2 Windows NT-4, 95, 98, ME, 2000, XP:

Klicken Sie auf den Start-Schalter Ihres Systemmenüs und wählen Sie "Ausführen". Starten Sie das Installationsprogramm "SETUP.EXE" auf der Installationsdiskette, bzw. -CD.

Das Installationsprogramm startet nun und fragt nach folgenden Informationen:

- Programmordner: "C:\PROGRA~1\WINCESY" ist hier die Vorgabe. Verwenden Sie Windows 95, so brauchen Sie nichts zu verändern. CESY unterstützt aus Kompatiblitätsgründen keine langen Dateinamen, daher sieht der Pfad so verkürzt aus. Falls Sie unter Windows 3.1 arbeiten, geben Sie hier das Verzeichnis ein, unter dem Sie CESY installieren möchten, z. B. "C:\WINCESY".
- Quelltexte-Verzeichnis. Hier werden die mitgelieferten Quelltexte, z. B. für das 8051-Systemprogramm, abgespeichert. Wenn Sie eigene Programme schreiben, werden auch diese zunächst dort abgelegt.
- Systemverzeichnis. Dorthin wird eine Datei, BWCC.DLL, kopiert, die zum Betrieb von CESY notwendig ist. Diese Datei gehört der Firma BORLAND, Inc., und wird von allen Programmen benutzt, die mit einem Compiler dieser Firma erstellt wurden. ACHTUNG! Von dieser Datei gibt es verschiedene Versionen! **Wenn Sie nicht die mitgelieferte Version benutzen, kann es vorkommen, dass CESY beim Compilieren abstürzt**.
- Name der Programmgruppe, in der die CESY-Dateien zusammengefasst werden sollen.

Zum Start der Installation klicken Sie bitte auf "Installieren". Darauf werden Sie noch gefragt, ob CESY die benötigten Verzeichnisse für Sie erstellen soll. Zuletzt öffnet sich ein weiteres Fenster, mit dem Sie das Kopieren der Dateien starten. Zusätzlich werden in diesem Fenster Statusinformationen zum Stand der Installation angezeigt.

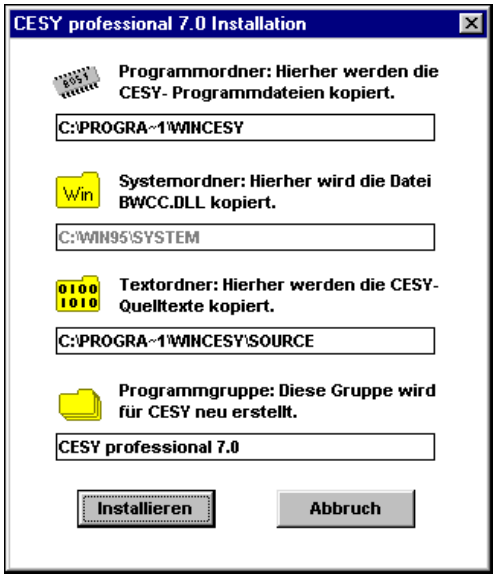


Abbildung 1: Installation

Als letztes erstellt SETUP die Programmgruppe für CESY. Hier finden Sie Verweise auf CESY und auf die CESY-Referenz (Online-Hilfe).

Die Installation ist damit abgeschlossen.

Je nachdem, welche Nutzer-Berechtigungen gesetzt sind, kann es vorkommen, dass einige Dateien nicht automatisch kopiert werden. Kopieren Sie gegebenenfalls dann selbst per Explorer oder DOS-Fenster die entsprechenden Dateien in den CESY-Ordner und ggf. die BWCC.DLL in den Windows-Systemordner.

2.1.3 Installation unter Virtual PC

CESY wurde unter dem Gesichtspunkt höchster Systemkompatibilität entwickelt und läuft daher auch unter Virtual PC auf verschiedenen Host-Systemen. Garantiert wird die Lauffähigkeit unter Apple OS X, da auf diesem System eine Vielzahl von Tests durchgeführt wurde. Hier ist insbesondere auf die Anbindung der seriellen Schnittstelle zu achten. Zielsysteme können problemlos über zugewiesene integrierte oder externe RS-232 Schnittstellen angesprochen werden. Gängige USB-RS232 Adapter funktionieren in der Regel gut, wenn die entsprechenden Treiber auch auf dem Gastsystem installiert werden.

2.1.4 Update-Installation

Installieren Sie ein eventuelles Update bitte grundsätzlich in einen neuen Ordner. Sie können im Nachhinein dann per Explorer die Ordner umbenennen. So wird vermieden, dass Ihre Quelltexte überschrieben werden.

2.2 Installation des C-Compilers

CESY stellt eine graphische Benutzeroberfläche für den Open-Source C-Compiler SDCC dar. SDCC steht für „Small Device C Compiler“, also C-Compiler für kleine Prozessoren. Die Referenz für SDCC liegt separat bei. Der SDCC ist komplett in CESY eingebunden, d.h. Sie können ein C-Programm genauso übersetzen wie ein Assembler- oder Basic-Programm. Der SDCC benötigt eine 32-bit Version von Windows, d.h. es läuft erst ab Windows 95.

Der Compiler muss separat installiert werden. Die Installation ist ganz einfach:

- Starten Sie das Installationsprogramm. Dieses heisst, je nach Version, etwa „SDCC_2_4_0_SETUP.EXE“.
- Die Installation erfolgt nun automatisch. Der Installationspfad sollte allerdings nicht, wie vorgegeben, im „Programme“-Ordner liegen, sondern etwa „C:\SDCC“. Dies erleichtert den Umgang, wenn SDCC über die Eingabeaufforderung aufgerufen wird.
- Für den Betrieb mit dem CESY-Monitorprogramm muss der Startup-Code abgeändert werden. Die geänderten Dateien sind in der Archiv-Datei „cesystartup.zip“. Kopieren Sie diese Datei in den SDCC-Ordner und entpacken Sie ihn dort. Es werden fünf Dateien, die alle mit „_startup“ anfangen, ersetzt.
- **Wichtig: Starten Sie den PC neu, auch wenn das Installationsprogramm Sie nicht dazu auffordert!**

Damit ist der SDCC-Compiler installiert. Wenn ein Teil des Tests nicht funktionieren sollte, konsultieren Sie die C-Dokumentation. Im Notfall hilft eine e-mail an uns. Rufen Sie dazu den Compiler von der Eingabeaufforderung auf wie folgt:

```
sdcc --print-search-dirs
```

und schicken Sie uns die ausgegebenen Meldungen.

2.3 Deinstallation

Vielleicht mögen Sie irgendwann nicht mehr mit CESY arbeiten und wollen das Programm von Ihrer Platte entfernen. Dazu ist nur der CESY-Ordner mit allen Programmdateien zu löschen. Außerdem sollten die Programmgruppen gelöscht werden. Die BWCC-Datei im SYSTEM-Ordner sollten Sie nicht löschen, da auch andere Programme darauf zugreifen könnten. Weitere Dateien werden von CESY nicht installiert.

Der SDCC-Compiler kommt mit einem eigenen Deinstallationsprogramm.

2.4 Starten von CESY

Starten Sie nun CESY mit einem Doppelklick auf das CESY-Symbol im Startmenü. Nach einer kurzen Information, die nur beim ersten Starten angezeigt wird, präsentiert sich CESY mit einem leeren Fenster und seinem Menü.

Innerhalb dieses Fensters werden Unterfenster, z. B. für Editoren, Monitor oder Terminal angezeigt. Sie können mit "Datei -- neu" diese Unterfenster aufrufen. Es ist möglich, beliebig viele Editor- und Monitorfenster zu öffnen, aber es kann nur ein Terminalfenster aufgerufen werden.

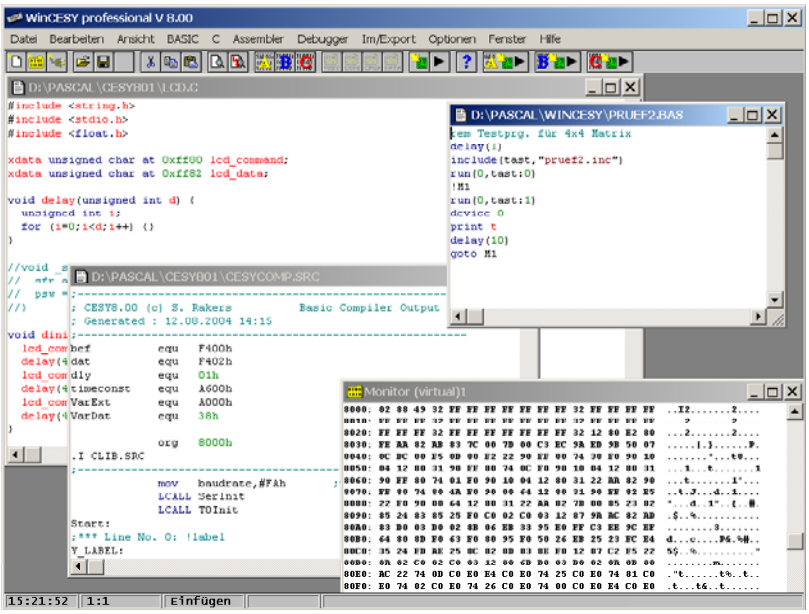


Abbildung 2: Das Programmfenster

2.5 Kennenlernen von CESY

Wenn Sie das erste Mal mit CESY arbeiten, lesen Sie jetzt bitte folgende Themen der Handbücher:

- Überblick
- Benutzeroberfläche
- Speicherkonzept
- Arbeitsschritte.
- Datei

Wir wollen nun zum weiteren Kennenlernen der wichtigsten Funktionen ein kleines Programm erstellen, das ein nicht-nachtriggerbares Monoflop emuliert. An einen Pin des Mikrocontrollers, sagen wir P1.0, wird ein Taster angeschlossen, der den Pin auf Masse zieht. Am Pin P1.7 soll eine Leuchtdiode angeschlossen werden, die nach den Tastendruck genau eine Sekunde aufleuchtet und dann wieder verlöscht.

Dieses Programm soll einmal in Assembler und einmal in BASIC implementiert werden.

Wenn Sie den Taster anschließen, bedenken Sie bitte, dass ein Pull-up-Widerstand vom Pin P1.0 nach +5V notwendig ist, sonst funktioniert das Beispiel nicht.

2.5.1 Assembler-Beispiel

Um die gewünschte Zeit genau einhalten zu können, ist die Benutzung eines Timers notwendig. Wir gehen davon aus, dass ein 8031 oder 8032 verwendet wird. Diese Bausteine unterscheiden sich nicht in der Timer-Programmierung. Sie können natürlich auch einen C515 oder C517 verwenden, denn die Beispiele funktionieren auch mit diesen Typen. Setzen Sie aber immer die .P31 Option.

Öffnen Sie nun also ein neues Editorfenster mit "Datei -- neu -- Editor". Unser Programm soll ab der Adresse 8000h im Speicher stehen. Die meisten Experimentier-Boards verfügen über einen RAM-Speicherbereich ab 8000h. Ist dies bei Ihrem Board anders, ersetzten Sie die 8000h bitte durch die benötigte Adresse. Die ersten Befehle lauten somit:

```
.P31
<tab>          org 8000h
```

<tab> bedeutet hier nur, dass Sie die Tabulatortaste drücken sollten, um den Befehl passend einzurücken. Die 8000h wird automatisch passend eingerückt. Der .P31 - Befehl sagt dem Assembler, dass er die SFR-Labels für den 8031-Prozessor voreinstellen soll. Sie könnten diese Einstellung auch über das Assembler-Menü vornehmen.

Als nächstes definieren wir die benötigten Labels:

```
tr0            bit    tcon.4
tf0            bit    tcon.5
reload         equ    55536
```

Beachten Sie, dass das Label "tcon" bereits vordefiniert ist, die Bit-Labels allerdings von Hand zu definieren sind.

Wir beginnen nun das Hauptprogramm. Zuerst ist der Timer zu initialisieren und der Ausgabepin zurückzusetzen:

```
               clr    p1.7
               clr    tr0
               clr    tf0
               anl    tmod,#11110000b
               orl    tmod,#00000001b
```

In einer Warteschleife soll das Programm verharren, bis die Taste gedrückt ist, danach wird der Ausgabepin geschaltet:

```
poll_key:      jb     p1.0,poll_key
               setb   p1.7
```

Jetzt wird eine Sekunde verzögert. Wir rufen dazu ein Unterprogramm, das 10 ms verzögert, hundertmal auf.

```
                mov   r0,#100
delay:          lcall wait_10ms
                djnz  r0,delay
                clr   p1.7
```

Der Ausgabepin ist nun ausgeschaltet. Wenn die Taste noch gedrückt ist, soll der Pin aber nicht wieder eingeschaltet werden. Daher muss noch ein weiteres Mal die Taste geprüft werden.

```
poll_key1:      jnb   p1.0,poll_key1
                sjmp  poll_key
```

Bis auf das Unterprogramm, das die eigentliche Verzögerung macht, ist unser Testprogramm fertig. Das Verzögerungs-Unterprogramm startet den Timer mit einem Startwert. Dieser Wert wird vom Timer in 1µs-Abständen inkrementiert (bei 12 Mhz Quarzfrequenz), Wenn ein Überlauf stattfindet, wird vom Controller das TF0-Flag gesetzt. Wir "pollen" also dieses Flag bis es gesetzt ist. Danach stoppen wir den Timer und kehren ins Hauptprogramm zurück.

```
wait_10ms:      mov   tl0,#.lo.reload
                mov   th0,#.hi.reload
                setb  tr0
poll_wait:      jnb   tf0,poll_wait
                clr   tr0
                clr   tf0
                ret
```

Noch eine Bemerkung zum Editor: wie Sie sicherlich schon bemerkt haben, rückt der Editor Ihre Programmzeilen automatisch ein. Das ist meistens höchst nützlich, kann aber auch nerven. Mit "STRG-RETURN" können Sie diese Eigenschaft abschalten. Ein zweiter Druck dieser Kombination schaltet das automatische Einrücken wieder an.

Das Programm ist nun fertig und sollte abgespeichert werden. Dazu rufen Sie bitte den "Datei -- Speichern unter" Befehl auf. Es öffnet sich ein Dialogfenster, in dem Sie den Dateinamen und den Dateityp angeben müssen. Bitte nennen Sie das Programm "ERST.SRC". "SRC" steht für "Source", engl. für Quelltext. Der Dateityp ist schon auf Quelltext voreingestellt.

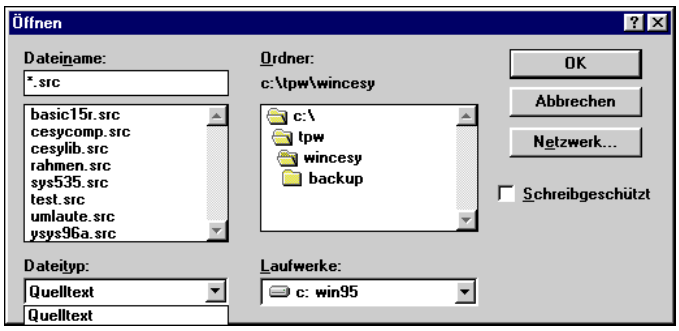


Abbildung 3: Datei-Dialog

Jetzt ist es an der Zeit, das Programm zu assemblieren. Das ist mit CESY ganz einfach. Mit dem Befehl "Assembler -- Assemblieren" startet der Assembler, übersetzt und linkt das Programm und schreibt es schließlich in den virtuellen Speicher von

CESY. Fertig. Am Ende bekommen Sie noch die Endadresse und die Anzahl der assemblierten Zeilen angezeigt. Wenn ein Fehler im Programm gefunden wurde, wird eine Fehlermeldung ausgegeben, und im Editor wird die entsprechende Zeile hervorgehoben dargestellt. Weitere Fehler können per Menübefehl angezeigt werden.

Mit dem Monitor schauen wir nun nach, ob das Programm auch im Speicher angekommen ist. Rufen Sie ihn bitte mit "Datei -- Neu -- Monitor" auf. Ein Monitorfenster erscheint, das einen Hexdump ab Adresse 0000h anzeigt. Mit dem Befehl "Bearbeiten -- Gehe zu" können Sie zur Adresse 8000h springen. Jetzt müsste ein Bild gemäß Abbildung 4 erscheinen. Anhand des Hexdumps kann man natürlich nicht auf die korrekte Funktion des Programms schließen. Aber zur Endkontrolle eignet sich diese Darstellung, da sie sehr kompakt ist und Anfang und Ende des Programms schnell zu erkennen sind. Man sieht zum Beispiel, dass das Programm nur ca. 48 byte lang ist. Genaue Informationen erhält man per Assembler-Listing.

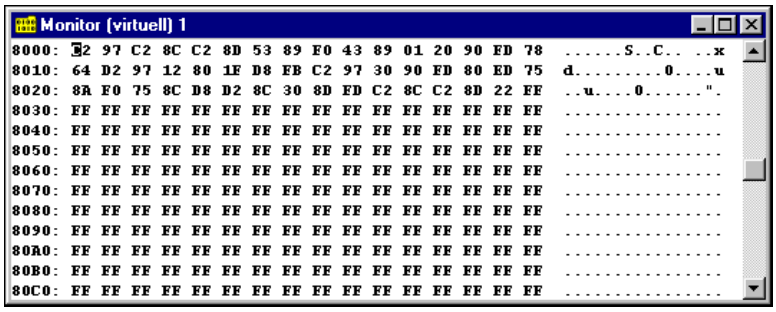

Monitor (virtuell) 1
8000: B2 97 C2 8C C2 8D 53 89 F0 43 89 01 20 90 ED 78 ......S..C.. ..x
8010: 64 D2 97 12 80 1F D8 EB C2 97 30 90 ED 80 ED 75 d.........0....u
8020: 8A F0 75 8C D8 D2 8C 30 8D ED C2 8C C2 8D 22 FF ..u....0......".
8030: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ....
8040: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ....
8050: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ....
8060: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ....
8070: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ....
8080: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ....
8090: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ....
80A0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ....
80B0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ....
80C0: FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF ....

Abbildung 4: Programm im Monitor

Speichern Sie nun Ihr Programm in binärer Form ab. Dies geschieht auch mit "Datei -- speichern unter". Wählen Sie aber nun als Dateityp "Binärdatei". Fast alle EPROM- oder FLASH-Programmiergeräte oder - Emulatoren verstehen dieses Datenformat. Wenn Sie ein EPROM brennen wollen, müssen Sie das Programm natürlich ab Startadresse 0000h assemblieren.

2.5.2 Download und Programmstart

Nun kann das Programm an das Zielsystem übertragen werden. Dazu sollte auf dem Zielsystem die CESY-Firmware laufen. Wie dieses angepasst wird, finden Sie in der Online-Hilfe im Kapitel "Das CESY-Betriebssystem". Dazu eine Anmerkung: Laden Sie einfach die benötigte Datei in den Editor, assemblieren Sie und schreiben Sie sich die Endadresse auf, die nach dem Assemblieren angezeigt wird. Speichern Sie dann das Programm in binärer Form ab. Startadresse ist 0000h, Endadresse wie aufgeschrieben. Alles wie bisher beschrieben. Programmieren Sie dann ein EPROM mit dem Firmware-Programm und setzen Sie es in Ihrem Mikrocontrollersystem ein. Nach dem Stromeinschalten und dem Verbinden mit dem PC sind Sie bereit für den Download Ihres gerade erstellten Programms.

Prüfen Sie bitte zunächst, ob das Zielsystem korrekt arbeitet und mit dem PC kommunizieren kann. Starten Sie dazu einen neuen Monitor ("Datei -- neu -- Monitor") und schalten Sie die Ansicht auf "XRAM" um. Jetzt sollte zügig ein Hexdump erscheinen. Das ist der Inhalt Ihres Firmware-EPROMs! Falls das Hexdump unvollständig oder sehr langsam erscheint, oder gar eine Fehlermeldung, klappt irgendet-

was mit der Verbindung nicht. Lesen Sie dazu das Kapitel "Hardware" der Online-Hilfe.

Nachdem die Verbindung steht, können Sie mit dem Download starten. Nach "Im/Export -- Download" erscheint ein Dialogfenster, in dem Sie den benötigten Adressbereich angeben müssen. Bitte runden Sie bei Startadressen auf volle "xx00h" und bei Endadressen auf volle "xxFFh". Das hilft, Probleme zu vermeiden. Die Zieladresse sollte mit der Startadresse identisch sein. Die Download-Adressen können übrigens auch automatisch eingesetzt werden. Das stellen Sie bei "Optionen -- Assembler" ein.

Starten Sie nun Ihr Programm mit "Im/Export -- Programm starten" und testen Sie es auf korrekte Funktion!

Wenn Sie bis hier gefolgt sind, wissen Sie wie man mit CESY schnell und einfach ein Programm editiert, assembliert, auf das Zielsystem lädt und startet. CESY kann jedoch noch viel mehr. Es stellt z. B. einen leistungsfähigen und komfortablen Debugger, einen Reassembler und einen BASIC-Compiler zur Verfügung. Letzter wird im nächsten Kapitel beschrieben, alle anderen Informationen finden Sie in der Online-Hilfe.

2.5.3 BASIC-Beispiel

Das gleiche Programm soll jetzt in BASIC geschrieben werden. Sie werden lernen, dass die Programmierung erheblich einfacher ist, jedoch auch erheblich mehr Ressourcen verbraucht werden. Wenn Sie beispielsweise die beliebten ATMEL-Controller mit eingebautem FLASH-PROM verwenden, stoßen Sie schnell an die Speicher-Obergrenze. Bei größeren Systemen mit "echten" 8031 oder 80535 o.ä. spielt es jedoch heutzutage keine Rolle mehr, ob Ihr Programm zwei oder 32 kB verbraucht - dann sollten Sie wirklich die Benutzung einer Hochsprache wie BASIC in Erwägung ziehen. Dank des Compilers und der schnellen 24-bit Integer Arithmetik sind gegenüber Assembler nur geringe Geschwindigkeitsnachteile zu erwarten.

Nun zum Programm. Zunächst ist wieder der Ausgabepin zurückzusetzen und der Timer zu initialisieren.

```
clear 144,7
openclk
```

Dann wird auf die gedrückte Taste an P1.0 gewartet:

```
!poll_1
ifs check(144,0)=1 goto poll_1
```

Der „ifs“-Befehl wirkt sich übrigens im Gegensatz zu „if“ nur auf die in der selben Zeile folgenden Kommandos aus.

Einschalten, verzögern und ausschalten:

```
set 144,7
delay(10)
clear 144,7
```

Insbesondere der "delay"-Befehl erlaubt eine sehr komfortable Timer-Programmierung, allerdings nur mit einer Auflösung von 100 ms. Das dürfte jedoch für die meisten BASIC-Programme ausreichen.

Jetzt noch darauf warten, dass die Taste wieder losgelassen wird, und dann wieder von vorne anfangen:

```
!poll_2
ifs check(144,0)=0 goto poll_2
goto poll_1
```

Fertig. Sie sollten das Programm nun abspeichern. Stellen Sie bitte den Dateityp auf "BASIC" ein. Wenn Sie das Programm das nächste Mal laden, wird das automatische Einrücken sofort deaktiviert.

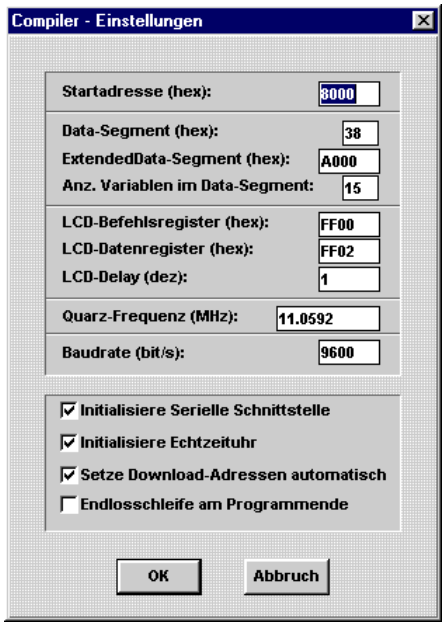


Abbildung 5: BASIC-Optionen

Das Compilieren des Programms erfordert einige Einstellungen, die im Dialogfenster "Optionen - BASIC" angegeben werden müssen. Die meisten Einstellungen sind nötig, da das BASIC ja Hardware-Funktionen unterstützt, die bei verschiedenen Controllerboards unterschiedlich sein können. Für dieses Programm ist insbesondere die richtige Einstellung der Quarz-Frequenz wichtig, denn diese bestimmt die Genauigkeit der Echtzeituhr. Da die Uhr im Programm explizit initialisiert wird, kann die automatische Initialisierung abgeschaltet werden. Außerdem müssen noch einige Einstellungen zur Speicherverwaltung gemacht werden. Stellen Sie als Startadresse 8000h ein. Diese Einstellung ist von höchster Wichtigkeit, da hinter der Startadresse sofort die Interrupt-Einsprünge liegen. Die Echtzeituhr arbeitet intensiv mit Interrupts. Wenn nun die Firmware die Interrupt-Einsprünge nicht an die richtige Adresse weitergibt, wird Ihr Programm sofort abstürzen. Die Firmware ist auf Adresse 8000h voreingestellt.

Wenn Sie unser „LAB-537“ verwenden, stellen Sie als Startadresse bitte „0000h“ ein, da dieser Bereich von der CESY-Firmware für Benutzerprogramme freigeschaltet ist. Die Interruptvektoren liegen dann auch auf jeden Fall richtig.

Die Einstellungen für Data- und ExtendedData-Segment brauchen Sie jetzt nicht, da keine Variablen verwendet werden. Genaueres dazu lesen Sie in der Online-Hilfe.

Starten Sie nun den Compiler. Eventuelle Programmfehler werden genauso angezeigt wie im Assembler. Wenn Sie die Endadresse beachten, werden Sie bemerken, dass Ihr Programm nun fast 4 kB lang ist! Das ist der Preis, der für die Benutzung einer Hochsprache zu zahlen ist. Das meiste ist allerdings die Standard-Bibliothek, die immer mitgeschleppt wird. Wenn Ihr Programm doppelt so lang wird, heißt das also nicht, dass auch der Speicherverbrauch entsprechend steigt.

Dafür ist das BASIC-Programm sehr viel übersichtlicher und schneller erstellt, da man nicht ständig nachschlagen muss, wie z. B. der Timer richtig initialisiert wird. Zum Ausprobieren des Programms muss es wie gehabt per Download ans Zielsystem geschickt und gestartet werden. Danach sollte es problemlos laufen.

2.5.4 C-Beispiel

Schließlich schreiben wir das gleiche Programm noch mal in C. Im Vergleich zu BASIC ist die Programmierung etwas schwieriger, da Befehle wie Delay nicht vordefiniert sind. Zu diesem Zweck benutzen wir hier einfach Leerschleifen. Die üblichen Compiler-Nachteile haben wir natürlich auch unter C, wie z.B. den höheren Resourcen-Verbrauch. C zahlt sich eigentlich erst aus, wenn die Programme komplex werden und eine Aufteilung in Bibliotheken erfordern. Diese Funktion stellt unser BASIC nicht bereit. Auch Interrupts sind in C einfacher zu programmieren.

Nun zum Programm. Zunächst definieren wir die Variablen und setzen den Ausgabepin auf null. Übrigens brauchen wir für diese einfache Programm kein „stdio.h“ oder weitere Bibliotheken!

```
sbit at 0x90 P1_0;
sbit at 0x97 P1_7;
int i;

void main(void) {
  while () {
    P1_7 = 0;
```

Dann wird auf die gedrückte Taste an P1.0 gewartet:

```
    while (P1_0 == 1) {};
```

Einschalten, verzögern und ausschalten:

```
    P1_7 = 1;
    for (i=0;i<10000;i++) {};
    P1_7 = 0;
```

Jetzt noch darauf warten, dass die Taste wieder losgelassen wird, und dann wieder von vorne anfangen:

```
    while (P1_0 == 0) {};
  }
}
```

Fertig. Sie sollten das Programm nun abspeichern. Stellen Sie bitte den Dateityp auf "C" ein. Wenn Sie das Programm das nächste Mal laden, wird das automatische Einrücken sofort deaktiviert.

Das Compilieren des Programms erfordert auch für den C-Compiler einige Einstellungen, die im Dialogfenster "Optionen - C" angegeben werden müssen. Die meis-

ten Einstellungen betreffen die Speicherverwaltung. Zumeist sind die Vorgaben ausreichend. Stellen Sie als Startadresse 0x8000 und als Datenadresse 0xC000 ein.

Wenn Sie unser „LAB-537“ verwenden, stellen Sie als Startadresse bitte „0000h“ ein, da dieser Bereich von der CESY-Firmware für Benutzerprogramme freigeschaltet ist. Die Interruptvektoren liegen dann auch auf jeden Fall richtig. Das XRAM-Segment sollte noch auf 8000h gesetzt werden.

Die Einstellungen für Data- und ExtendedData-Segment können Sie einfach auf „Auto“ belassen.

Starten Sie nun den Compiler. Sollten Programmfehler auftreten, so wird ein Fenster geöffnet, in dem die Ausgabe-Logdatei angezeigt wird. Fehler werden in roter Farbe angezeigt, Warnungen in orange und Mitteilungen in grün. Klicken auf eine Fehlermeldung oder eine Warnung, in der eine Zeilennummer angegeben wird, führt direkt ins Editorfenster auf die betroffene Zeile.

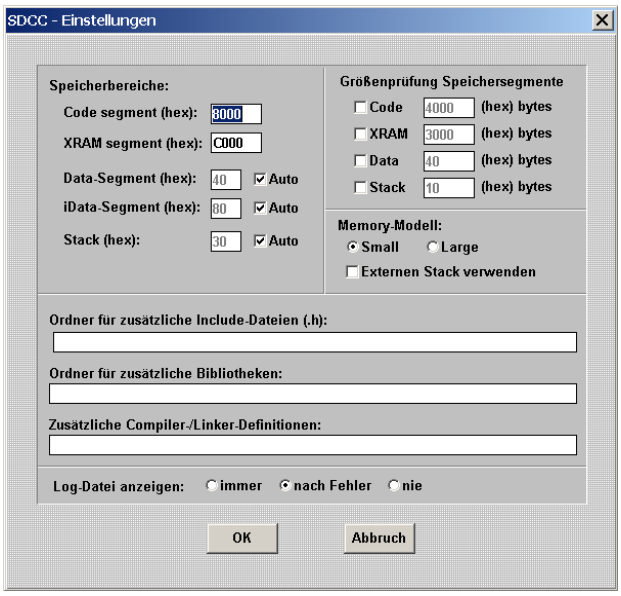


Abbildung 6: C-Compiler-Optionen

Eventuelle Programmfehler werden hier genauso angezeigt wie im Assembler. Wenn Sie die Endadresse beachten, werden Sie bemerken, dass Ihr Programm auch in C etwa 4 kB lang ist. Das meiste ist allerdings auch hier die Standard-Bibliothek, die immer mitgeschleppt wird. Wenn Ihr Programm doppelt so lang wird, heißt das also nicht, dass auch der Speicherverbrauch entsprechend steigt.

Zum Ausprobieren des Programms muss es wie gehabt per Download ans Zielsystem geschickt und gestartet werden. Danach sollte es problemlos laufen.

2.6 Probleme erkennen und lösen

Oft gibt es bei der ersten Benutzung von CESY ein paar Probleme. Die üblichen Fehlerquellen seien hier kurz aufgelistet.

- Download und Programmstart funktionieren nicht.

  ⇒Drücken Sie die RESET-Taste Ihres Zielsystems, öffnen Sie ein Monitorfenster und stellen Sie die Ansicht auf "XRAM". Jetzt sollte ein Hexdump zügig erscheinen. Wenn dies nicht der Fall ist, stimmt etwas mit der Firmware oder der Übertragung nicht. Beachten Sie, dass für höhere Baudraten als 19200 ein einigermaßen aktueller Rechner (P-III 1GHz o.ä.) erforderlich ist.

- Im Monitorprogramm funktioniert die Ansicht von XRAM, CROM oder Data nicht.

  ⇒Ist ein funktionsfähiges Betriebssystem vorhanden?

  ⇒Sind die Übertragungsparameter in der CESY-Firmware richtig (insbesondere Quarzfrequenz)?

  ⇒Benutzen Sie einen Quarz, der präzise Baudraten ermöglicht (z. B. 11,0592 MHz, oder 12 MHz bei Systemen mit Baudratengenerator)?

  ⇒Funktioniert Ihr serielles Kabel? Dies kann man leicht prüfen: schalten Sie die Stromversorgung Ihres Zielsystems ab und entnehmen Sie den Controllerbaustein. Verbinden Sie die RxD- und die TxD- Leitung des leeren Controllersockels (8031: Pins 10 und 11) miteinander und schalten Sie den Strom wieder ein. Öffnen Sie ein Terminalfenster und geben Sie einige Zeichen ein. Diese sollten nun doppelt erscheinen, wenn die Pins verbunden sind, ansonsten einfach. Ist das nicht der Fall, überprüfen Sie bitte Ihr Kabel. Dieser Standard-Test sollte in der Anleitung Ihres Boards genau beschrieben sein.

- Assemblerprogramme funktionieren, aber BASIC-Programme wollen nicht laufen.

  ⇒Steht Ihr Programm an einer Einsprungadresse, die in der Firmware für die Interruptquellen festgelegt ist (Label "D_start")?

  ⇒Haben Sie den Controllertyp (8031/32, 80535, 80537) im Menü richtig eingestellt? Wenn Sie einen anderen Controller verwenden, sollten Sie 8031 oder 8032 einstellen. Eventuell müssen Sie in der Datei "CLIB51.SRC" Änderungen vornehmen.

  ⇒Verzichten Sie auf die Benutzung und Initialisierung der Uhr und der seriellen Schnittstelle. So werden keine internen Hardwareresourcen des Controllers benutzt. Wenn das funktioniert, nehmen Sie die Peripherie schrittweise wieder in Betrieb.

  ⇒Steht die Datei "CLIB.SRC" im selben Verzeichnis wie "CESY.EXE"? Wenn dies nicht der Fall ist, kann es sehr seltsame Fehlermeldungen geben. Kopieren Sie die Dateien dann an die richtige Stelle.

2.7 Programme in ein EPROM brennen

Vielleicht möchten Sie Ihre soeben erstellten Programme in ein EPROM brennen. Dazu müssen Sie sowohl beim Assembler-, als auch beim BASIC-Programm die Startadresse auf 0000h festlegen. Übersetzen Sie Ihr Programm neu und speichern Sie es binär ab. Das Binärformat sollte jeder EPROM- oder FLASH-Programmer lesen können. Ansonsten steht noch das Intel-Hex-Format zur Verfügung.

3 CESY-Referenz

Dieses Kapitel gibt eine detailliertere Einführung in die Bedienung und Konzepte von CESY. Insbesondere wird der Monitor und die Arbeit mit dem Assembler und Debugger behandelt. Die BASIC- und C-Compiler sind Gegenstand der folgenden zwei Kapitel.

3.1 Die Benutzeroberfläche

Die CESY-Oberfläche fasst alle benötigten Werkzeuge zusammen. Beliebig viele Editor- und Monitorfenster können gleichzeitig geöffnet werden. Dagegen ist das Öffnen nur eines Terminalfensters möglich. Wie in allen Windows-Programmen können die einzelnen Fenster verschoben, maximiert und minimiert werden. Bei Bedarf öffnen sich weitere PopUp-Fenster, z. B. für den Debugger oder Einstellungen. Menüoptionen, die gerade unmöglich sind, werden schattiert dargestellt. Mit der Toolbar können die wichtigsten Funktionen direkt angesprungen werden.

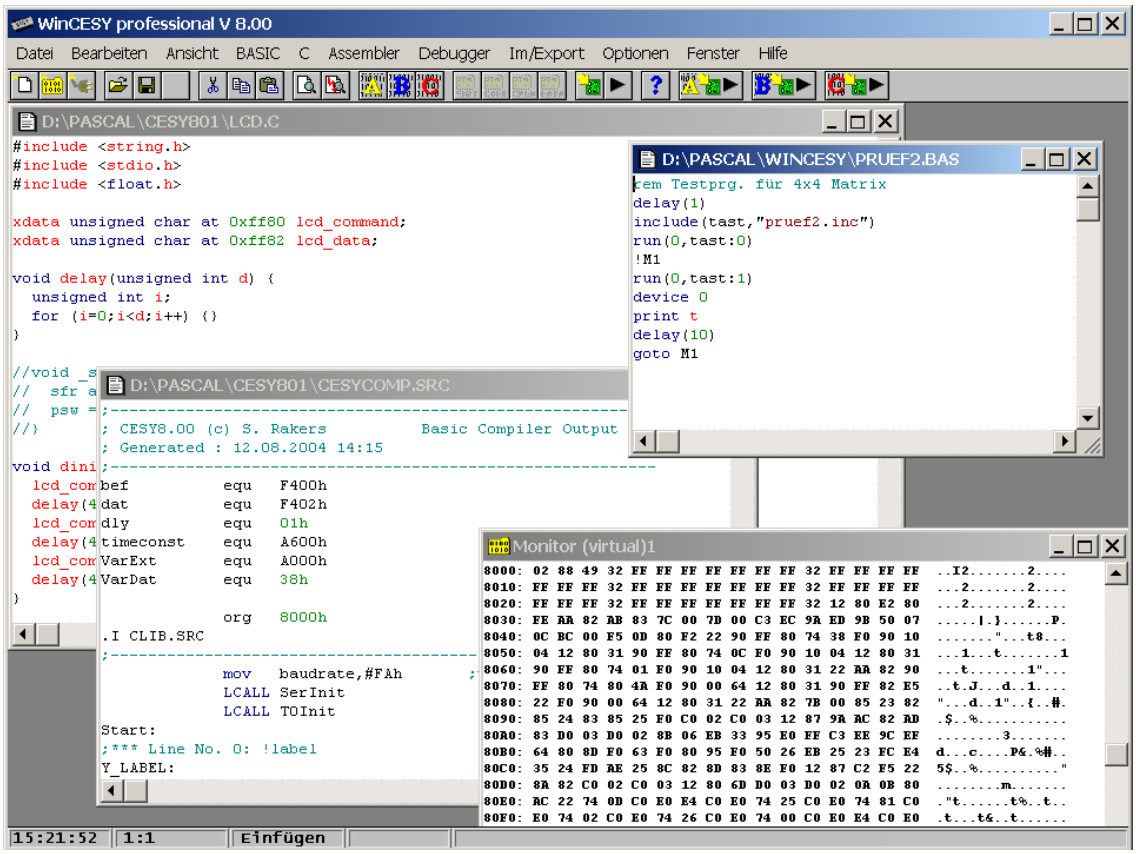


Abbildung 7: Die CESY-Oberfläche

3.2 Das Speicherkonzept

Das wichtigste Merkmal von CESY ist der sogenannte *Virtuelle Codespeicher*. Als "virtuell" bezeichnet man etwas, das in Wirklichkeit gar nicht vorhanden ist, dem Benutzer aber vorgegaukelt wird, so dass er es sehen und sogar damit arbeiten kann. CESY gaukelt Ihnen also den Speicher eines Controllersystems vor, auch wenn gar keines angeschlossen ist. Dieser Speicher nimmt den vom Assembler generierten Code auf. Sie können ihn mit dem Monitor bearbeiten. Sein Inhalt kann ganz oder häppchenweise über die Download- oder Upload-Funktionen zwischen dem virtuellen Speicher auf dem PC und den Speicherbänken des Zielsystems übertragen werden. Diskettenoperationen sind nur auf den virtuellen Speicher möglich. So müssen Sie zunächst einen Upload durchführen und können dann den Speicherbereich auf einen Datenträger abspeichern, wenn Sie z. B. den RAM-Inhalt Ihres Controllersystems speichern möchten.

Der Speicher eines 80x51-Systems ist in drei Speicherbänke aufgeteilt, die übereinander liegen und daher mit verschiedenen Befehlen angesprochen werden müssen:

• *DATA-Segment*: auch interner Datenspeicher. Umfasst beim 8031/51 128 byte, beim 8032/52 und 80535 256 byte. Dieser Speicher liegt direkt im Controllerchip, daher ist der Zugriff sehr schnell. Dieser Speicher enthält auch dern Prozessorstack. Wenn Sie mit der Benutzung dieses Speichers auskommen, benötigen Sie keinen externen RAM-Chip.

• *CODE-Segment*: auch Code-ROM. Dieser Speicher kann als EPROM oder Masken-ROM im Chip integriert sein, meistens wird jedoch ein externer EPROM-Chip benutzt. Dort liegt der ausführbare Programmcode.

• *XRAM-Segment*: auch externer Datenspeicher. Befindet sich in einem externen RAM-Chip und kann bis zu 64 kB umfassen. Aus diesem Speicher können ohne Tricks keine Programme ausgeführt werden, er dient allein der Speicherung von Daten. Benötigt man einen Download zum Testen von Programmen, so muss hardwaremäßig eine kleine Manipulation vorgenommen werden.

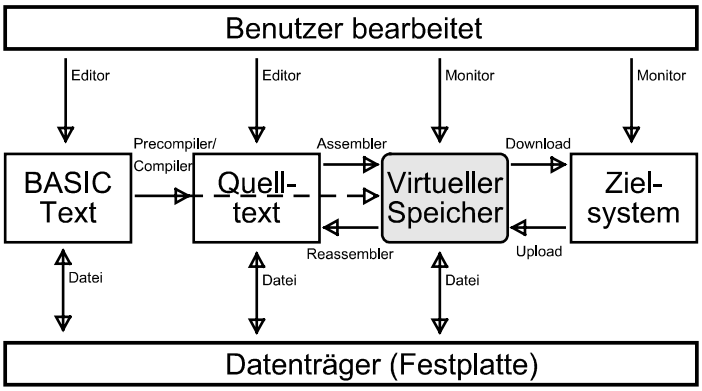


Abbildung 8: Speichermanagement

3.3 Vom Quelltext zum lauffähigen Programm

Um ein Programm zu erstellen und gleich auch auszuführen sind nur wenige Schritte notwendig, gesetzt den Fall, alles funktioniert auf Anhieb.

1. Neues Editorfenster öffnen
2. Assembler- oder BASIC-Programm im Editor editieren
3. Sichern der Quelltext-Datei
4. Assemblieren, bzw. Compilieren
5. Sichern als (Save As) Binär- oder Hexdatei
6. Download
7. Starten

3.4 Das Datei-Menü

Über das Datei-Menü wird, wie bei Windows-Programmen üblich, die Kommunikation des Programmes mit der Peripherie, also Datenträgern und Drucker, geregelt. Eine Ausnahme bildet die serielle Schnittstelle, sie wird vom Debugger, Monitor und Terminalprogramm zur Laufzeit verwaltet.

• *Neu...* öffnet ein neues Editor-, Monitor- oder Terminalfenster. Für letztere ist dies die einzige Möglichkeit aktiv zu werden. Sie können beliebig viele Editor- und Monitorfenster, aber nur ein einziges Terminal- oder Projektfenster öffnen.

• *Öffnen...* dient zum Öffnen einer Textdatei, also Quelltext, BASIC- oder C-Programm, und zum Laden von Binär- oder Intel-Hex-Dateien in den virtuellen Speicher. Abhängig von der Wahl des Dateityps wird ein Editorfenster geöffnet und der Text angezeigt, oder der Code in den virtuellen Speicher geladen.

• *Geschützt Öffnen...* öffnet ein geschütztes Editorfenster, in dem nicht editiert werden kann. Dient auch zum Kontrollieren von Intel-Hex-Dateien, denn diese werden hier als Textdatei geöffnet und nicht in den virtuellen Speicher geschrieben.

• *Speichern* speichert den Text im aktiven Fenster. Nicht zum Speichern von Binärcode!

• *Speichern unter...* speichert je nach gewähltem Dateityp Text-, Hex- oder Binärdaten. Bei letzteren öffnet sich ein Dialogfenster, in dem nach Start- und Endadresse gefragt wird.

• *Drucken...* druckt den Quelltext oder einen Speicherbereich aus dem virtuellen Speicher aus, je nach dem ob der Befehl bei aktivem Editor- oder Monitorfenster aufgerufen wird. Bein Ausdrucken werden Zeilennummern und Zeilenschattierungen mitgedruckt. *Drucken von Text*: Wurde vorher der Quelltext assembliert, kann in das Listing der generierte Opcode eingebunden werden (Knopf "*Assembler-Info*"). Dies ist äußerst nützlich, da man nun zu jeder Zeile die Adresse im Code zu-

ordnen kann. Wahlweise kann der ganze Text oder nur der markierte Teil (Knopf *Nur Markierung*) gedruckt werden.

• *Drucken eines Hexdumps*: Der Hexdump wird entweder mit 16 oder mit 32 Byte pro Zeile ausgedruckt.

Es ist nicht möglich, aus dem Datei-Menü ein C-Projekt zu speichern, da dies bei jeder Änderung ohnehin automatisch erfolgt.

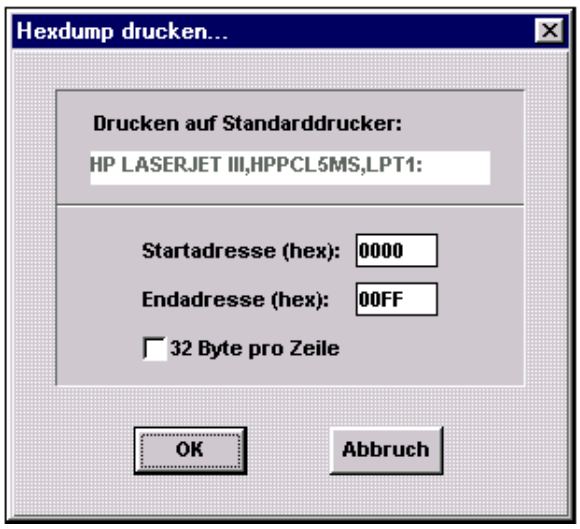


Abbildung 9: Hexdump drucken

3.5 Der Editor

Sie können ein Editorfenster im Menü "**Datei**" mit den Optionen "*Neu -- Editor*" oder "*Öffnen*" öffnen. Bei ersterem Befehl wird die AutoFormat-Funktion automatisch gesetzt, bei letzterem Befehl nur, wenn die geöffnete Datei vom Typ ".src" oder ".inc" ist.

Der Editor dient nicht nur zum Eingeben Ihres Programmes, sondern auch zur Anzeige der Position von gemeldeten Programmfehlern, zum Setzen von Breakpoints, sowie zur Anzeige der aktuellen Programmzeile im Debugger.

Wenn Sie zum Erstellen Ihrer Programme lieber einen etwas komfortabler ausgestatteten Editor verwenden möchten, so steht dem nichts im Wege - er sollte nur, wie auch der CESY-Editor, reinen ASCII-Code erzeugen.

Der CESY-Editor unterstützt Sie bei der Eingabe eines Assembler-Quelltextes ganz besonders, indem er die Eingaben automatisch formatiert. Die Funktion ist bei der Eingabe eines BASIC- oder C-Programmes natürlich eher hinderlich und lässt sich daher mit *STRG+Return* abschalten.

Der *Editor* ist intuitiv bedienbar. Die Funktionen im einzelnen:

- Cursorsteuerung mit den *Cursortasten*.
    - Wortweise links/rechts mit *STRG-Cursor links/rechts*.
    - *Pos1*, *Ende* befördert den Cursor ans linke, bzw. rechte Zeilenende.
    - *Bild auf/ab* blättert eine Seite nach oben/unten.
    - *STRG+Pos1*, Ende bringt den Cursor an den Anfang, bzw. das Ende des Textes.
    - *TAB* springt zur nächsten Tabulator-Position.
- Einfügen und Löschen wie gewohnt mit *Entf*- und *Einfg*-Tasten
    - *STRG+Y* löscht die Zeile, in der der Cursor steht.

  - *Einfg* schaltet zwischen Einfüge- und Überschreibmodus um.
  - *STRG+Return* schaltet die automatische Formatierung ein- oder aus.

- Textblöcke werden mit den Cursortasten bei *gehaltener Shift*-Taste oder mit der Maus markiert. Mit Hilfe der Zwischenablage können Sie Textausschnitte zwischen verschiedenen Anwendungen oder zwischen verschiedenen Editor-Fenstern transferieren. Im **Bearbeiten**-Menü stehen folgende Blockoperationen zur Verfügung:
  - *Ausschneiden*. Der Block wird in die Zwischenablage kopiert und danach aus dem Text entfernt.
  - *Kopieren*. Der Block wird in die Zwischenablage kopiert.
  - *Einfügen*. Ein Block aus der Zwischenablage wird in den Text eingefügt.
  - *Löschen*. Der Block wird gelöscht, ohne die Zwischenablage zu beeinflussen.
- *Suchen und Ersetzen*: Nach der Anwahl dieser Menüpunkte öffnet sich ein Dialogfenster, indem Sie den Suchbegriff und den Ersetzungstest, sowie Suchoptionen angeben können. Mit der Option "*Weitersuchen*" führen Sie eine Suche nach dem ersten Auffinden eines Suchbegriffes fort.
  - *Auffinden von Fehlerstellen*: Der Assembler merkt sich die ersten 10 Fehlerstellen und zeigt direkt nach dem Assemblieren die erste an. Mit der Option "*Fehler suchen*" springen Sie zum nächsten Fehler. Die Meldung wird in der Statuszeile angezeigt.
  - *Direkter Sprung auf eine Zeile*: Große Programme sind oft langwierig zu durchblättern. Daher kann eine Zeile durch Angabe der Zeilennummer automatisch angesprungen werden.

Für die bessere Übersicht über den Programmcode gibt es eine farbliche Syntax-Hervorhebung. Diese erleichtert das Eingeben von Programmen, da Schlüsselworte, bzw. Mnemonics farblich gekennzeichnet werden. Sie können die Farben mit dem Menübefehl „Optionen – Textfarben“ auswählen (nebenstehende Abbildung). Die Syntax-Hervorhebung ist auf älteren Rechnern recht langsam. Sie kann jedoch ebenfalls im Textfarben-Menü abgeschaltet werden.

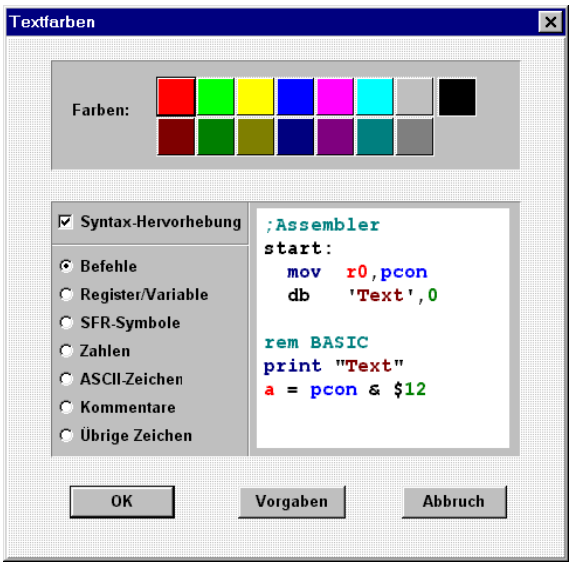


Abbildung 10: Farbauswahl

3.5.1 Geschützter Editor

Dateien, die vor versehentlicher Änderung geschützt werden sollen, können im Dateimenü mit der Option "*Geschützt öffnen*" geöffnet werden. In dem nun geöffneten Editorfenster kann der Text zwar angezeigt, aber nicht verändert werden. Öffnet man hier eine Hex-Datei, so wird nicht der Code in den virtuellen Speicher geladen, sondern man kann den Dateiinhalt einsehen.

3.6 Der Assembler

Das **Assembler**-Menü enthält die Befehle zum Assemblieren und Reassemblieren, zum Anzeigen der Labeldefinitionen sowie zur Einstellung des Controllertyps.

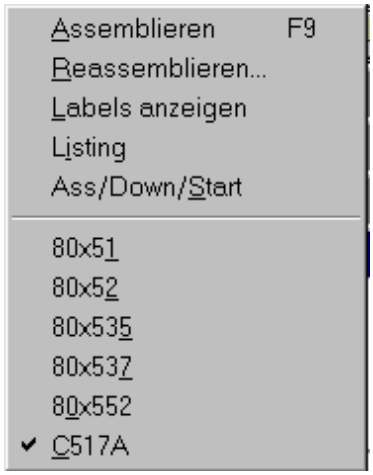


Abbildung 11: Assembler-Menü

Sobald in diesem Menü der Befehl "*Assemblieren*" angewählt wird, beginnt der Assemblerlauf. In einem Fenster wird die Nummer der gerade assemblierten Zeile angezeigt. Außerdem werden bei der Assemblierung gefundene Fehler mitgezählt. Die ersten zehn Fehlermeldungen werden mit Position gespeichert. Sie können im Editor angesprungen werden. Natürlich können unsinnige Fehlermeldungen dadurch entstehen, dass der Assembler nach einem Fehler nicht wieder gut aufsetzt. Wenn z.B. bei einer Labeldefinition ein Fehler auftrat, werden alle Befehle, die auf dieses Label Bezug nehmen, auch als Fehler erkannt und gespeichert.

Nach dem Assemblieren steht der erzeugte Code im virtuellen Speicher. Sie können ihn als Hexdump mit dem Monitor einsehen. Beachten Sie bitte, dass der Code noch nicht in einer Datei steht oder sich bereits im Zielsystem befindet. Benutzen Sie zu diesem Zweck den Download-Befehl.

Die erzeugten Labels werden automatisch in eine Datei des Typs ".LAB" geschrieben. Diese können Sie mit dem Befehl "*Labels anzeigen*" in einen geschützten Editor laden.

Mit dem Menüpunkt "*Listing*" erzeugen Sie eine Datei, die den Quellcode zusammen mit dem generierten Objektcode übersichtlich enthält. Solche Dateien sind zur Dokumentation eines Projektes oder zur Fehlersuche sinnvoll. Die generierte Datei wird gleich in einem neuen geschützten Edit-Fenster geöffnet. Dieser Befehl ist übrigens nur für Assembler-Programme gedacht und funktioniert nicht mit BASIC. Eingebundene Include-Dateien werden ebenfalls nicht mitausgegeben.

Die Prozessor-Einstellungen dienen zur Wahl der vordefinierten Labels. Sie müssen eine dieser Optionen verwenden, können aber selbstverständlich weitere Labels definieren, falls Sie einen anderen Controller verwenden als die hier möglichen Einstellungen. Nehmen Sie dann den 80x51 als Grundtyp und definieren Sie die Labels für die Erweiterungen.

Die Prozessor-Einstellungen sind auch für den BASIC-Compiler gültig und müssen in jedem Fall in diesem Menü vorgenommen werden!

3.7 Assembler-Syntax

3.7.1 Quelltextformat

Zeilen dürfen beliebig eingerückt werden. Benutzen Sie dafür den Tabulator im Editor - er ist genau passend für 80x51-Befehle eingerichtet. Groß- und Kleinschreibung wird ignoriert, soweit die Zeichenkette nicht quotiert, d.h. in Anführungszeichen eingeschlossen ist. Kommentare können an jede Zeile angehängt werden. Sie werden mit einem Semikolon gekennzeichnet. Kommentare dürfen keine einfachen Anführungszeichen enthalten. Die letzte Zeile im Text sollte eine Leerzeile sein.

3.7.2 Label-Verarbeitung

Kompatibilität mit dem ASM51 von Intel erfordert vier Typen von Labels: DATA, EQU, BIT und CODE-Labels mit verschiedenen Typen dürfen gleiche Namen haben; bitte vermeiden Sie dieses aber, denn einige Befehle arbeiten mit mehr als einem Labeltyp. Verwechslungen sind dann vorprogrammiert.

Von den Labels werden die ersten 12 Zeichen ausgewertet. Das dürfte den meisten Verwechslungen vorbeugen.

- **CODE**: Alle Labels, die als Sprungmarken im Quelltext stehen, sind automatisch vom Typ CODE. Man kann Labels auch als Code definieren. Sprünge und "MOV DPTR,#" arbeiten mit CODE.
- **DATA**: Befehle, die den internen Datenspeicher ansprechen, verlangen den Labeltyp DATA. Die SFR-Labels sind alle vom Typ DATA.
- **BIT**: Bitverarbeitungsbefehle verlangen den Typ BIT. Das adressierte Bit wird mit einem Punkt gekennzeichnet. Falls in einer BIT-Zuweisung ein Label ge-

braucht wird, muss dieses vom Typ DATA und vorher definiert sein. Nicht alle Adressen sind Bit-adressierbar, sondern nur 20h - 2fh und 80h, 88h ... F8h.

- **EQU**: Befehle, die eine Konstante erwarten, benötigen den EQU- Typen. Die Konstante darf 8bit oder 16bit breit sein. Einige Operatoren wie .LO. oder .HI. arbeiten nur mit Konstanten.

Grundsätzlich gilt für die Typen BIT, EQU und DATA, dass ein Label vor den Zugriff definiert sein muss. Das zwingt den Programmierer, alle Definitionen am Anfang des Programms zu tätigen. Beim Typ CODE wäre das freilich nicht sinnvoll.

Beispiel zu den Label-Typen:

```
weg      CODE     2000h
zaehler  DATA     50h
TI       BIT      SCON.1           ;SFR-Label
RI       BIT      98h.0
test     BIT      10h.5            ;!Fehler!
konst8   EQU      8
konst16  EQU      1600h
;- - - - - - - - - - - - - - - - - - - - -
Marke:   JZ       weg              ;CODE-Beispiele
         LJMP     Marke
         MOV      DPTR,#weg
         MOV      DPTR,#Marke
         MOV      A,zaehler        ;DATA-Beispiele
         MOV      A,#zaehler       ;!Fehler!
         MOV      C,TI             ;BIT-Beispiele
         MOV      A,#konst8        ;EQU-Beispiele
         MOV      DPTR,#konst16
```

3.7.3 Zahlensysteme und Arithmetik

Ungekennzeichnete Zahlen werden als Dezimalzahlen interpretiert. Hexadezimal- und Binärzahlen werden mit einem nachgestellten "h", bzw. "b" markiert. Ascii-Zeichen dürfen mit Ausnahme von Steuerzeichen wie Komma, Klammer, Doppelkreuz etc. verwendet werden, sie sind dann in einfache Anführungszeichen zu setzen.

Es dürfen beliebige arithmetische Ausdrücke verwendet werden. Falls das Ergebnis negativ ist, wird es ins Zweierkomplement umgewandelt. Klammern sind erlaubt, ansonsten gilt die Reihenfolge der Eingabe. Es stehen die Operatoren **- + * / .DIV. .MOD. .AND. .OR. .XOR. .LO. .HI. .NOT.** zur Verfügung. Division durch 0 er-

gibt 0. Operatoren aus Buchstaben werden von Punkten eingeschlossen, damit sich keine Einschränkungen für die Labels ergeben.

Beispiel zur Arithmetik:

```
        MOV   A,#20         ;dezimal
        MOV   A,#'a'        ;ASCII klein
        MOV   A,#20h        ;hex
        MOV   A,#10101111b;binär
        MOV   A,#-20        ;negativ
ma:     MOV   DPTR,#ma-3    ;Arithmetik
        MOV   A,#.LO.ma
```

3.7.4 Pseudo-Befehle

Für die Steuerung des Assemblers stehen folgende Pseudo-Befehle zur Verfügung:

- **ORG**: Anfang des Codes im Speicher
- **DB**: Tabelle aus Bytes. Darf bis zu 64 Elemente enthalten. Texte sind auch erlaubt und mit Bytes mischbar. Außerdem Konstanten.
- **DW**: Tabelle aus Words. Darf bis zu 32 Elemente enthalten. Labels der Typen CODE und EQU sind erlaubt.
- **DS**: Leerraum im Code. Angabe in Bytes.
- **.P**: Definition des Controllertyps. Die hier angewendete Definition hat eine höhere Priorität als die Definition im Assembler-Menü. Es sind die Befehle .P 51, .P 52, .P 535, .P537, .P552 möglich. Das Programm kann diese Einstellungen über Symbole erkennen, die für die bedingte Assemblierung automatisch global definiert werden. Beachten Sie bitte, dass dieser Befehl gleich am Zeilenanfang stehen muss und zwischen dem großen P und der Zahl genau ein Leerzeichen zu stehen hat.

Beispiel zu den Pseudo-Befehlen:

```
ORG 2000h                   ;Startadresse
.P 535                      ;Prozessor ist 80535
DB  12,23h,konst8           ;Bytes
DB  'Text',0dh,0ah,0        ;Text
DW  1480,13000,AFFEh        ;Words
DW  Unter1, Unter2, Unter3 ;Sprungtabelle
DS  33                      ;33 Byte freier Platz
```

3.7.5 Include-Dateien

Wenn der Quelltext zu lang wird, so dass er unübersichtlich wird, oder wenn für einige Anwendungen Bibliotheken erstellt werden sollen, sollten Teile in eine Datei ausgelagert werden. Auch Makrobibliotheken können (und müssen) als Include-Dateien realisiert werden.

- **.I <*Dateiname*>**

Der Befehl muss direkt am Zeilenanfang stehen, sonst gibt es einen Syntax-Fehler! Pfade sollten nicht angegeben werden. Sorgen Sie also dafür, dass alle Include-Dateien im Projekt-Verzeichnis stehen! - Zwischen dem Befehl und dem Dateinamen muss genau ein Leerzeichen stehen. Sie dürfen Include-Dateien bis zu 10 Ebenen tief schachteln, d.h. eine Datei bindet die andere ein, die wiederum eine dritte einbindet usw. Um dieses Feature zu nutzen, müssen Sie bei den Assembler-Optionen das Kontrollfeld „rekursive Include-Dateien“ aktivieren.

Beispiel:

Sieht der Quelltext folgendermaßen aus:

```
            RET
 .I UNTER.SRC
            LCALL Unterprg ; Label aus Include-Datei
            ANL   A,#3     ; Ergebnis verarbeiten
```

und existiert eine Datei namens "UNTER.SRC", in der das Label "Unterprg" definiert ist:

```
Unterprg:   MOV   A,P1
            ORL   A,#55h
            RET
```

so liest der Assembler den Quelltext folgendermaßen:

```
            RET
 Unterprg:  MOV   A,P1
            ORL   A,#55h
            RET
            LCALL Unterprg ; Label aus Include-Datei
            ANL   A,#3     ; Ergebnis verarbeiten
```

Bitte beachten Sie, dass ein Include-Befehl niemals der letzte Befehl im Text sein sollte. Hängen Sie gegebenenfalls ein NOP an.

- **.L <Dateiname>**

Dieser Befehl funktioniert genau wie der .I-Befehl, mit dem Unterschied, dass die Include-Datei lokal ist. Sie darf keine Programmteile aus anderen Programmen aufrufen. Außerdem sind die Labels dieser Datei für das aufrufende Programm unsichtbar. Dies gibt dem Programmierer eine größere Freiheit in der Wahl der Labels. Der Preis ist, dass die in der Include-Datei enthaltenen Programme nur über eine Sprungtabelle aufrufbar sind, da ja keine Labels nach außen sichtbar sind. Wird aus einer lokalen Datei eine globale Datei eingebunden, sind deren Labels und Symbole in der einbindenden Datei sichtbar, nicht jedoch eine Stufe oberhalb.

3.7.6 Makros

Ein Makro ist eine im Allgemeinen kurze, im Quelltext oftmals wiederkehrende Befehlsfolge. Diese wird einmal definiert und kann vielmals wieder aufgerufen werden. Es können Parameter angegeben werden. Diese werden beim Aufruf einfach ersetzt, so dass auch Parameter wie Register "R0" möglich sind, die ja als Label verboten wären. In einem Makro dürfen auch Labels vorkommen. Sie sind lokal; wird innerhalb des Makros ein Label aufgerufen, so wird zunächst versucht, eine lokale Definition, also innerhalb des Makros, zu finden. Erst dann wird nach draußen gesprungen. Ein Makro darf nie ein anderes Makro aufrufen!!!

- Definition: **MAK** ***Name(Par1,Par2...Par8)***

In der Definition dürfen außer nach dem MAK keine Leerzeichen vorkommen. Die Parameter dürfen bis zu 6 Zeichen lang sein.

- Ende der Definition: **MAEND**
- Aufruf: ***Name() oder Name*****(Par1,Par2...)**

Beispiel:

Der Befehl INC DPTR wird häufig verwendet; ein Befehl wie DEC DPTR fehlt jedoch im Befehlssatz. Abhilfe schafft ein Makro:

```
       MAK  DECDPTR
       DEC  DPL
       XCH  A,DPL
       CJNE A,#ffh,end
       DEC  DPH
end:   XCH  A,DPL
       MAEND
;-----------------------------
       ORG  1000h
       MOV  DPTR,#1000h
       DECDPTR()
       RET
```

Beachten Sie bitte, dass, falls keine Parameter benötigt werden, beim Aufruf dennoch Klammern zu setzen sind. Ansonsten wird das Makro nicht erkannt.

3.7.7 Bedingte Assemblierung

CESY bietet eine einfache Art, bedingt zu assemblieren. Dazu können Symbole definiert werden, deren Existenz mit einem IF-Konstrukt abgefragt werden können. Die Symbole stehen in keinem Zusammenhang mit Labels o. ä., sondern stehen vollkommen für sich.

Eine Definition wird mit *$define oder $gldefine* gemacht. Der Unterschied zwischen den Befehlen ist, dass bei *$define* das Symbol nur unterhalb der Definition im Quelltext definiert ist, während sich *$gldefine* unabhängig von der Position auf den gesamten Text auswirkt. Mit einer *$if .. $else .. $endif* - Konstruktion lassen sich die Symbole auswerten. Symbole sind Case-sensitiv, das heißt, Sie müssen auf Groß- und Kleinschreibung achten.

Beispiel:

```
$define Reg3
       MOV  A,#0
$if Reg3
       MOV  R3,A
$else
       MOV  R0,a
$endif
       RET
```

Dieses Beispiel wird im Assembler zu:

```
        MOV   A,#0
        MOV   R3,A
        RET
```

Ferner geben die automatisch erzeugten Symbole %PRxxx% den eingestellten Prozessortypen an. Diese Symbole gelten ab der Stelle, an der ein .P - Kommando gefunden wird. Findet der Assembler kein solches Kommando, so wird ein globales Symbol definiert, das dem per Menü selektierten Prozessortypen entspricht.

Beispiel:

```
.P 537
        MOV   A,#0
$if %PR537%
        MOV   R3,A
$else
        MOV   R0,A
$endif
        RET
```

Bemerkungen:

- Schachtelung ist nicht erlaubt
- $else kann weggelassen werden
- Befehle sind klein zu schreiben
- In INCLUDE-Dateien bitte nur globale Deklarationen!

3.7.8 Fehlermeldungen des Assemblers

Der Assembler beschränkt sich auf wenige, eindeutige Fehlermeldungen. Nach dem Auftreten eines Fehlers und Bestätigung durch den Benutzer wird im Editor automatisch die Fehlerstelle angesprungen.

Nicht immer ist ein Fehler auch dort zu finden, wo er auftritt. Besondere Vorsicht lassen Sie bitte bei mehr als einer Fehlermeldung walten! Es kann sein, dass der Assembler nach einem Fehler nicht wieder gut aufsetzt, und sich mit Fehlermeldungen beschwert, obwohl es dazu keinen Grund gibt. Die Fehlermeldung "Label nicht gefunden" kann auch bei einem Tippfehler auftreten, z.B. wenn eine Hexadezimalzahl nicht gekennzeichnet wurde.

Wenn ein Fehler beim Einbinden eines Makros auftritt, kann nur die Stelle der Einbinde-Anweisung angezeigt werden. Der Fehler ist dann irgendwo im Makro zu finden. Hier ist eine Fehlersuche oftmals schwierig, denn man sieht nur schwer, wenn lokale und globale Labels durcheinanderkommen.

Wenn Sie Include-Dateien benutzen, werden auch die Fehler, die dort auftreten, angezeigt. Es erscheint ein Fenster, in dem die ersten 10 gefundenen Fehler jeweils mit Zeilennummer und Namen der Datei, in der sie aufgetreten sind, angezeigt werden. In Dateien, die bereits geöffnet sind, werden die fehlerhaften Zeilen automatisch markiert. Sie können von der Fehlerliste aus die entsprechenden Fenster anklicken. Falls die Datei noch nicht geöffnet war, können Sie sie auch von dem Fehlerlisten-Fenster aus öffnen.

Die Fehlerliste lässt sich auch vom "Bearbeiten"-Menü aus anzeigen. Bedenken Sie aber, dass jede Stelle nur einmal angesprungen werden kann, obwohl sie weiterhin in der Fehlerliste erscheint. Es ist möglich, eine Datei ein zweites Mal zu öffnen, obwohl bereits ein entsprechendes Fenster offen ist.

3.7.9 Die einzelnen Fehlermeldungen:

- ***Label mehrfach deklariert***. Labels sind Konstanten und können nicht überschrieben werden.
- ***Syntax - Fehler*** Der Befehl wurde nicht erkannt.
- ***Fehler bei Pseudo-Befehl*** Der Pseudo-Befehl wurde nicht erkannt, oder falsch angewendet.
- ***Speicherüberlauf*** Diese Fehlermeldung kann bei Systemen mit weniger als 16 MB Speicherplatz auftreten. Dann hilft nur eins: aufrüsten.
- ***Label nicht gefunden*** Das aufgerufene Labels wurde nicht deklariert. Beachten Sie, dass Konstanten im Quelltext oberhalb des Aufrufes definiert werden müssen.
- ***Bit falsch adressiert*** Nur Speicherstellen im internen Datenspeicher von 20h - 2fh und 80h, 88h ..f8h sind bitadressierbar.
- ***Sprungbereich überschritten*** Der SJMP-Befehl und die bedingten Sprungbefehle erlauben nur einen Sprungbereich von +/- 127 Byte.
- ***Falsche Adressierungsart*** Diese Adressierungsart ist für den verwendeten Befehl nicht gültig.
- ***Blockübergang bei AJMP/CALL*** Diese Befehle sind nur in einem 2k-Block gültig. Blockübergänge sind nicht möglich.
- ***Nicht abgeschlossener Text*** Ein Text in einer DB-Anweisung muss an beiden Seiten mit einfachen Anführungszeichen (') abgeschlossen werden.
- ***Falsche Labeldefinition*** Das Label ist ungültig. Ein Label muss immer mit einem Buchstaben beginnen und darf nur aus Buchstaben, Ziffern und dem _-Zeichen bestehen.
- ***Pseudo-Befehl zu lang*** Pseudo-Befehle dürfen eine Länge von 32 Elementen nicht überschreiten.

- ***Makro nicht definiert*** Das Makro ist in dieser Form/Schreibweise nicht definiert worden. Wie bei Labels gilt, dass die Definition im Quelltext oberhalt des Aufrufes stehen muss.
- ***Parameterzahl falsch*** Das Makro muss mit exakt der gleichen Zahl an Parametern aufgerufen werden wie es definiert wurde.

3.7.10 Bemerkung zum SDCC-Compiler:

Der SDCC-Compiler erzeugt natürlich aus dem C-Code erst einmal ein Assemblerprogramm, das in einem zweiten Schritt übersetzt wird. Dieser intermediäre Code ist nie in CESY sichtbar, steht aber in den erzeugten Dateien. SDCC benutzt zum Übersetzen dieses Codes einen anderen Assembler als den CESY-Assembler, so dass hier Inkompatibilitäten auftreten. Wenn Sie sich einfach nicht um den Zwischencode kümmern, werden Sie keine Probleme haben.

3.8 Der Reassembler

Zu einem Crossassembler-Entwicklungssystem sollte immer auch ein Reassembler gehören, denn häufig befindet man sich in der Situation, dass man ein Programm hat, von dem kein Quelltext (mehr) vorhanden ist. CESY besitzt einen komfortablen und schnellen Reassembler, der sich auch von Tabellen und ASCII-Zeichenfolgen nicht aus dem Konzept bringen lässt. Der Reassembler arbeitet 100% symbolisch, das heißt, absolute und relative Sprünge sowie Ladezugriffe auf Tabellen werden automatisch an Sprung- und Zieladresse mit Labels versehen. Auch Sprünge, die nicht auf das erste Byte eines Befehls springen, werden erkannt. Das Label wird vor den Befehl gesetzt und dem Aufruf wird einfach eine Konstante addiert. Zugriffe auf Tabellen werden immer auf den Anfang der Tabelle umgerechnet, auch hier wird im Quelltext zum Label eine Konstante addiert. Außerdem werden Zugriffe auf SFR-Register immer per Label adressiert.

Leider ist es für die 80x51-Sprache nicht möglich, eine automatische Tabellenerkennung zu programmieren, da jeder Code von 0 bis FF einen Befehl darstellt. Sie müssen also wissen, wo sich Tabellen befinden. Schauen Sie sich im Monitor dazu den Code Ihres Programms genau an! Logische Zahlenfolgen, ASCII-Zeichenfolgen, die Wörter bilden oder unsinnige Befehlsfolgen weisen meist auf eine Tabelle hin.

Auch der Reassembler wird durch den aktiven Prozessor-Modus beeinflusst. Je nach Modus werden die passenden SFR-Labels erzeugt.

3.8.1 Bedienungsschritte:

- Laden Sie den Programmcode in den Codespeicher. Meist liegt der Code als Hexcode oder Binärcode vor. Ist letzteres der Fall, so ist es unbedingt erforderlich, dass die richtige Startadresse angegeben wird, da sonst viele Sprünge nicht symbolisch adressiert werden können.
- Sehen Sie im Monitor nach, ob Sie Tabellen finden können und notieren Sie gegebenenfalls Start- und Endadresse und ob die Tabelle als Bytefolge oder als ASCII-Zeichen im Programmtext erscheinen sollen.
- Wählen Sie im Assembler-Menü den Reassembler-Befehl.
- Es erscheint ein Fenster, in dem Sie Tabellen und Textbereiche erfassen können. Geben Sie in den zugehörigen Eingabefeldern Start- und Endadresse an und klicken Sie auf den entsprechenden "Hinzufügen"-Knopf. Die Informationen werden automatisch sortiert. Achten Sie bitte darauf, dass es keine Überschneidungen gibt.

- Tragen Sie danach Start- und Endadresse des zu reassemblierenden Codes ein.
- Wählen Sie, ob der Code symbolisch reassembliert werden soll und ob der erzeugte Text in ein neues Editorfenster oder in eine Datei geschrieben werden soll.

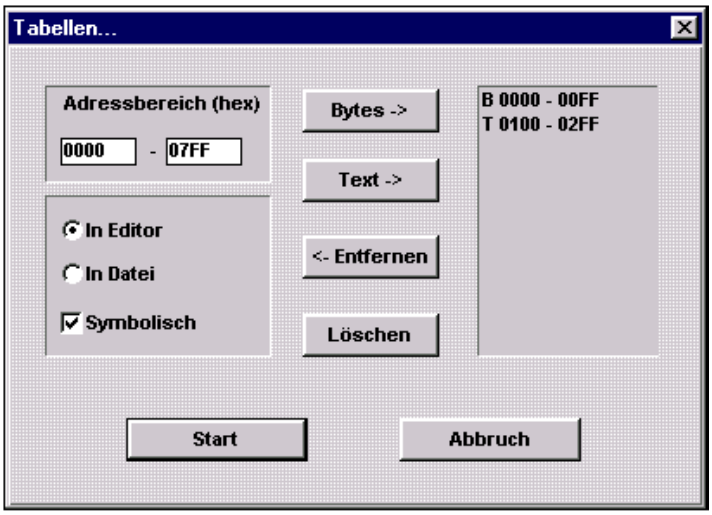


Abbildung 12: Tabellen-Editor des Reassemblers

3.8.2 Tipps:

• Möglicherweise müssen Sie mehrfach versuchen, Ihren Code zu reassemblieren. Ihre Antworten werden jeweils zwischengespeichert, so dass beim nächsten Mal das Fenster schon die nötigen Parameter enthält.

• Symbolisch reassemblieren sollten Sie immer, denn im entstehenden Text lässt sich nicht verfolgen, wo ein Sprung wie "LJMP 146Fh" tatsächlich hinzielt.

• Wenn der entstehende Programmtext so lang wird, dass er nicht mehr in den Editorspeicher passt (über 12000 Zeilen), können Sie ihn in eine Datei schreiben lassen, die Sie mit einem anderen Editor bearbeiten können. Sie können diese Datei auch als Include-Datei assemblieren lassen.

• beim Reassemblieren kann nur eine Fehlermeldung auftreten: Speicherüberlauf. Dabei wird der Reassemblerlauf abgebrochen. Sie sollten dann den Bereich verkleinern. Reassemblieren Sie nicht mehr als 8 kB auf einmal. Schon aus 8 kB Code werden schnell mehrere Tausend Zeilen Quelltext.

3.8.3 Zu erwartende Probleme

Wenn am Ende des Programms eine Tabelle steht, diese aber dem CESY nicht als Tabelle bekanntgemacht wurde, und in dieser Tabelle ein relativer Sprungbefehl steht, der auf eine Adresse außerhalb des Programms steht, so zeigt der Sprung auf ein Label, das jedoch nicht deklariert wird (wie auch, wenn es außerhalb des Programms stehen müsste). Eine eventuelle Assemblierung des erzeugten Textes würde natürlich einen Fehler anzeigen.

Es ist durchaus normal, wenn sich der erhaltene Quelltext nicht auf Anhieb assemblieren lässt. Eine Überarbeitung ist meistens unumgänglich.

3.9 Der Monitor

Überblick über die 64 kByte virtuellen Speicher und, wenn Sie den PC mit einem 80x51-System verbunden haben, auch über die Speicherbänke des Zielsystems, verschafft Ihnen der Monitor. Er wird aus dem Dateimenü mit dem Befehl "Neu -- Monitor" aufgerufen.

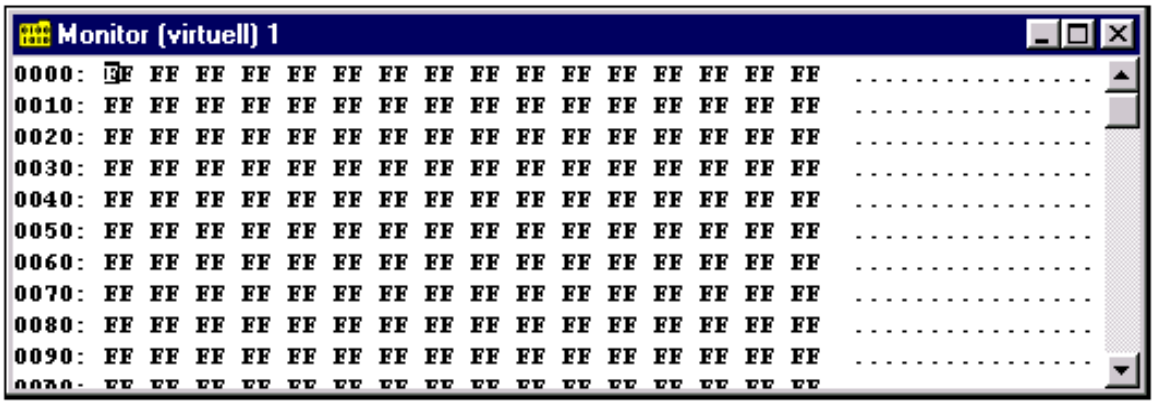


Abbildung 13: Monitor

Sie sollten den Monitor gut beherrschen, denn er ist das mächtigste Instrument des Entwicklungssystems, mit dem Sie die Ergebnisse aller anderen Funktionen kontrollieren und zusammenfügen können!

Der Monitor stellt in seinem Fenster einen Speicherauszug als Hexdump dar. Sie können den Cursor mit der Maus auf eine beliebige Stelle setzen und auch mit den Cursortasten im Hexdump bewegen. Mit der Tabulatortaste wechseln Sie von der Hex-Darstellung zur ASCII-Darstellung. Durch die Eingabe eines Hex-Wertes oder eines Zeichens editieren Sie den Speicher.

Über das **Ansicht**-Menü stellen Sie ein, ob Sie den virtuellen Speicher oder eine der drei Speicherbänke (Data, XRAM, Code) im Zielsystem bearbeiten. Es ist möglich, mehrere Monitorfenster mit jeweils verschiedenen Ansichten gleichzeitig zu benutzen. Beachten Sie bitte, dass die Code-Speicherbank nicht beschrieben werden kann und es daher beim Editierversuch eine Fehlermeldung gibt.

Damit der Zugriff auf das Zielsystem möglich ist, muss es über die serielle Schnittstelle an den PC angeschlossen sein, und das CESY-Betriebssystem muss laufen. Näheres hierzu im Kapitel "Hardware" des Hilfesystems. Falls aus irgendeinem Grund die Verbindung zum Zielsystem zusammenbricht, wird eine Fehlermeldung "Timeout" angezeigt.

Im "**Bearbeiten**"-Menü finden Sie die notwendigen Befehle, um mit dem Monitor den Speicher zu verwalten. Sie können Speicherbereiche, die jeweils in einem Dialogfenster einzugeben sind, in die Zwischenablage kopieren, aus der Zwischenabla-

ge einfügen oder mit einem bestimmten Wert füllen, und Sie können nach einer Bytefolge in einem Adressbereich suchen.

Wenn ein Monitorfenster aktiv ist, können Sie mit dem Befehl "Datei -- Drucken" einen Speicherbereich ausdrucken.

3.9.1 Tipps

Sie können über die Zwischenablage einen Speicherbereich im Zielsystem von einer Speicherbank in die andere übertragen. Stellen Sie dazu zuerst die Ansicht der Quell-Bank, z. B. "Code" ein, wählen Sie den Befehl "Bearbeiten -- Kopieren" und kopieren den benötigten Bereich in die Zwischenablage. Dann stellen Sie die Ansicht auf die Ziel-Bank, z. B. "XRAM" und fügen den Bereich aus der Zwischenablage an der gewünschten Adresse ein.

Beim Einfügen wird natürlich nicht richtig eingefügt, sondern der Bereich überschrieben.

Wenn Sie einen Speicherbereich des Zielsystems auf einen Datenträger speichern möchten, müssen Sie zunächst im Im/Export-Menü einen Upload vornehmen, um den Bereich in den virtuellen Speicher zu holen. Nur daraus ist ein Speichern auf einen Datenträger oder das Ausdrucken möglich.

Wenn Sie die Anzeige in disassemblierter Form vermissen: durch die Möglichkeit der Anzeige mehrerer Editorfenster ist es bequem möglich, zwischendurch den benötigten Bereich zu reassemblieren. Danach schließen Sie das nicht mehr benötigte Editorfenster einfach wieder.

3.9.2 Troubleshooting

Der Monitor teilt sich mit den anderen Programmteilen, insbesondere mit dem Terminal, eventuell eine serielle Schnittstelle. Falls Probleme mit der Übertragung auftreten, sollten alle Monitorfenster und das Terminalfenster geschlossen, das Zielsystem zurückgesetzt und das Monitorfenster erneut in der gewünschten Ansicht geöffnet werden.

Bei jedem Scrollen wird das Monitorfenster neu aufgebaut. Ist eine andere Ansicht als die des virtuellen Speichers gewählt, so kann der Neuaufbau unter Umständen einige Zeit dauern. Eine schnelle Verbindung zum Zielsystem, mindestens 4800 bit/s, ist wünschenswert.

3.10 Der Debugger

Für die Funktionen des Debuggers ist das CESY-Systemprogramm im Zielsystem erforderlich, dessen Beschreibung Sie im Abschnitt "Hardware" der Online-Hilfe finden. Außerdem muss die Leitung INT0 auf Masse gezogen werden, damit der Debugger nach jedem Programmschritt einen Statusbericht vom Zielsystem erhält. Auch dies ist genau im Hardware-Kapitel nachzulesen.

Der Debugger ist ausschließlich für das Testen von Assemblerprogrammen gedacht. BASIC-Programme können auch getestet werden, wenn man sie zunächst mit dem Präcompiler übersetzt und dann assembliert. Man bleibt aber in jedem Fall auf Assembler-Ebene.

C-Programme können zurzeit nicht im Debugger behandelt werden. Dies ist jedoch in Arbeit.

3.10.1 Debugger-Menü

Das Debugger-Manü enthält drei Menüpunkte:

- *Assembler*
- *Source-Level*
- *Breakpoint*

Der erste Menüpunkt ist zum Debuggen von Programmen vorgesehen, von denen kein Quelltext vorliegt oder die nicht in der aktuellen Sitzung assembliert und zum Zielsystem übertragen wurden. Hier wird nur der jeweils als nächstes auszuführende Befehl angezeigt.

Der zweite Menüpunkt dagegen ruft den Debugger als Source-Level-Debugger auf. Dazu muss des zum Code gehörende Editorfenster geöffnet und der Quelltext assembliert worden sein. Der Vorteil dieses Debug-Modus ist, dass die betreffende Zeile im Editor farblich gekennzeichnet wird, so dass sich das Programm auf Quelltextebene verfolgen lässt.

Schließlich gibt es die Möglichkeit, Breakpoints zu setzen. Dieser Befehl wird aus dem Editorfenster aufgerufen.

3.10.2 Benutzung

Im Folgenden wird der Debugger im Source-Level Modus erklärt. Der Assembler-Modus funktioniert mit Ausnahme der Quelltextanzeige genauso.

Das Debug-Fenster ist in zwei Bereiche aufgeteilt. In dem linken Debug-Bereich werden die Register A, B, R0-R7, DPTR, SP, ST, PC sowie der nächste Befehl angezeigt und können auch verändert werden. Es gibt folgende Funktionen, die durch die entsprechenden Knöpfe aufgerufen werden:

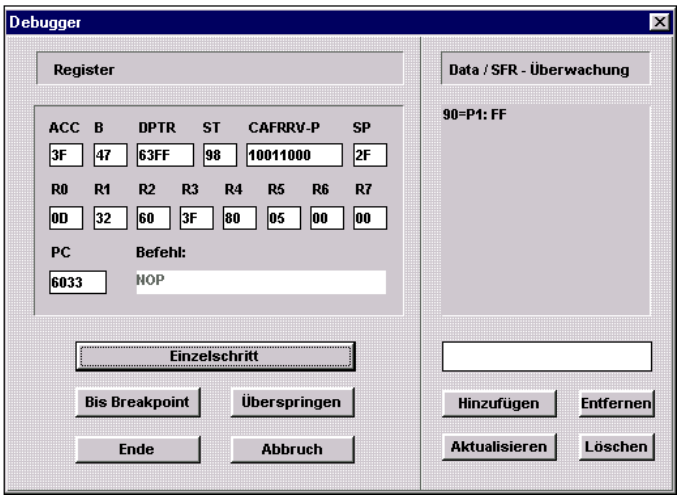


Abbildung 14: Debugger-Fenster

- *Einzelschritt* führt den Programmschritt ab der im Eingabefeld "PC" angegebenen Adresse mit Übergabe aller Register aus und kehrt danach zum Debugger

zurück. Alle Register-Felder werden aktualisiert.

• *Bis Breakpoint* startet ein Programm mit Übergabe aller Register und kehrt erst nach Erreichen eines Breakpoints zum Debugger zurück. Die Funktion kann, falls das Programm den Breakpoint nicht erreicht, mit der Funktion "Abbruch" abgebrochen werden. Eine Weiterarbeit ist erst nach Verlassen des Debuggers und anschließendem Reset des Zielsystems möglich.

• *Überspringen* führt den nächsten Befehl aus. Falls dieser Befehl ein Unterprogrammaufruf war, wird das Unterprogramm komplett abgearbeitet, erst danach wird das Debugger-Fenster aktualisiert, und Sie können weiterarbeiten. Die Funktion kann, falls das Programm den Breakpoint nicht erreicht, mit der Funktion "Abbruch" abgebrochen werden. Eine Weiterarbeit ist erst nach Verlassen des Debuggers und anschließendem Reset des Zielsystems möglich.

• *Abbruch* bricht die Funktionen "Bis Breakpoint" und "Überspringen" ab und kehrt zum Debugger-Fenster zurück. Eine Weiterarbeit ist erst nach Verlassen des Debuggers und anschließendem Reset des Zielsystems möglich.

• *Ende* beendet den Debugger.

3.10.3 Watches

Der rechte Teil des Fensters ist für die "Watches", d. h. Überwachung von SFR- und Datenspeicherzellen zuständig. In einer Listbox werden die bereits definierten Adressen angezeigt. Nach jedem Einzelschritt oder Erreichen eines Breakpoints zeigt der Debugger nicht nur alle Register neu an, sondern aktualisiert auch die Listbox.

•*Hinzufügen* fügt eine im zugehörigen Eingabefeld angegebene Adresse zur Watch-Liste hinzu. Wenn die Adresse im Bereich von 0 bis 7Fh liegt, wird angenommen, dass es sich um eine Adresse im Datenspeicher handelt, andernfalls wird die Adresse als SFR-Register interpretiert. Die Adresse kann komfortabel als Label angegeben werden, in der Liste erscheint sie dann zusätzlich in hexadezimaler Form.

•*Entfernen* entfernt den Markierten Eintrag aus der Liste.

•*Löschen* löscht alle Einträge in der Liste.

•*Aktualisieren* aktualisiert die Einträger der Liste.

3.10.4 Tipps

Der Debugger benötigt den externen Interrupt INT0. Falls Ihr Programm diesen auch benötigt, ist kein Arbeiten mit dem Debugger möglich, wie überhaupt auftretende Interrupts den Debugger stören.

Sie können den Debugger auch dazu benutzen, ein Programm mit definierter Registerübergabe in Echtzeit zu starten. Dazu muss die INT0-Leitung von Masse genommen werden. Das Programm wird dann mit dem Knopf "Einzelschritt" gestartet. Wenn Ihr Programm mit dem RET Befehl endet, kehrt es zum Debugger zurück, und die Register und Watches werden angezeigt. Das Programm sollte nicht länger

als einige Millisekunden brauchen, sonst meldet der Debugger einen Timeout-Fehler.

Zeitverhalten: Der Debugger arbeitet nicht in Echtzeit. Durch das ständig mitlaufende Protokoll zum PC wird Ihr Programm um einen Faktor in der Größenordnung 1000 langsamer. Zeitkritische Programme können daher nicht debuggt werden.

Natürlich benötigt der Debugger auch die serielle Schnittstelle, über die er mit CESY in Verbindung steht. Das zu debuggende Programm sollte also auch nicht auf die Schnittstelle zugreifen. Diese Einschränkungen sind in der Praxis normalerweise von untergeordneter Bedeutung, da eher Algorithmen als ganze Programme getestet werden.

3.10.5 Troubleshooting:

Der Debugger teilt sich mit den anderen Programmteilen eventuell eine serielle Schnittstelle. Falls Probleme mit der Übertragung auftreten, sollten alle Monitorfenster, das Terminalfenster und das Debugfenster geschlossen, das Zeilsystem zurückgesetzt und das Debugfenster erneut geöffnet werden.

Bedenken Sie, dass der Text im Assembler mit einer leeren Zeile enden sollte. Ansonsten kann eine Fehlermeldung wie "Bitte assemblieren Sie erst Ihren Quelltext" auftreten, wenn nach dem letzten Befehl im Text (auch wenn das ein Sprungbefehl ist) die Adresse einer weiteren Zeile gesucht wird.

Erscheint beim Debuggen ein Timeout-Fehler, obwohl der Download und der Monitor in XRAM-Ansicht funktionieren, so erhöhen Sie das Timeout-Limit unter „Optionen – I/O-Parameter“, z.B. auf einen Wert von 30-50. Dieser Wert gibt die Wartezeit an, die CESY auf die Rückmeldung vom Board wartet. Leider hängt diese Zeit von der Arbeitsgeschwindigkeit Ihres Rechners ab. Bei schnellen PCs kann eine Erhöhung auf ca. 100 notwendig sein!

3.11 Kommunikation mit dem Zielsystem

Das **Import/Export**-Menü besitzt folgende Befehle:

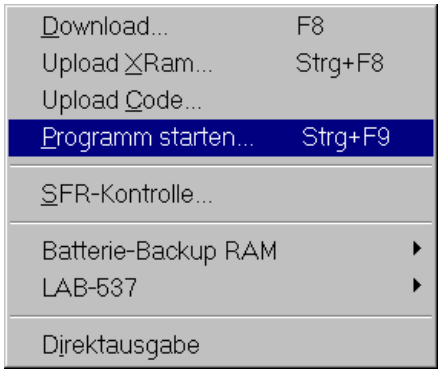


Abbildung 15: Import/Export Menü

• *Download* kopiert einen Speicherbereich vom virtuellen Speicher in den XRAM-Speicher des Zielsystems. Die Start- und Endadresse sowie die Startadresse im Zielsystem müssen in einem erscheinenden Pop-Up Fenster angegeben werden. Für die Übertragung ist ein Zielsystem mit CESY-Betriebssystem erforderlich.

• *Upload XRAM* kopiert einen Speicherbreich vom XRAM-Speicher des Zielsystems in den virtuellen Speicher. Die Start- und Endadresse im Zielsystem sowie die Startadresse im virtuellen Speicher müssen in einem erscheinenden Pop-Up Fenster angegeben werden. Für die Übertragung ist ein Zielsystem mit CESY-Betriebssystem erforderlich.

• *Upload CROM* kopiert einen Speicherbreich vom Code-ROM-Speicher des Zielsystems in den virtuellen Speicher. Die Start- und Endadresse im Zielsystem sowie die Startadresse im virtuellen Speicher müssen in einem erscheinenden Pop-Up Fenster angegeben werden. Für die Übertragung ist ein Zielsystem mit CESY-Betriebssystem erforderlich.

•*Programm starten* startet ein Programm im Zielsystem. Wenn das Programm nur kurze Zeit (unter 50 ms) läuft, so wird nach dessen Ende mit dem RET-Befehl ein Fenster angezeigt, in dem die Register des Controllers wie sie sich nach dem Programm ergeben, angezeigt. Beachten Sie bitte, dass ein Programm nur im Code-ROM laufen kann, und daher ein Programm-Download eigentlich nicht möglich ist. Ein üblicher Kunstgriff zur Umgehung dieses Problems wird im Hardware-Abschnitt beschrieben.

Abbildung 16: SFR-Fenster

•*SFR-Kontrolle* ermöglicht das Auslesen und Setzen von SFR-Registern des Controllers. In einem Pop-Up Fenster kann die auszulesende oder zu ändernde Adresse eingegeben werden. Wenn der Knopf "Einlesen" gedrückt wird, wird der Inhalt des Registers gelesen und angezeigt. Ist die Checkbox "permanent auslesen" aktiviert, so wird die Adresse permanent ausgelesen und angezeigt, bis der "Stop"-

oder "Ende"-Knopf angeklickt wird. Um ein Register zu setzen, ist ein Wert in das "Inhalt"-Feld einzutragen und der "Ausgeben"-Knopf anzuklicken.

- *Batterie-Backup RAM* enthält zwei Untermenüs:

  - *Autostart setzen* setzt eine Markierung in der Nähe der Interrupt-Vektoren Ihres Programms. Wird ein Reset ausgelöst, so prüft das CESY-Betriebssystem, ob diese Markierung vorhanden ist und startet dann das Programm. Diese Funktion mach nur Sinn, wenn das Zielsystem mit einem nicht-flüchtigen Programmspeicher ausgestattet ist.

  ACHTUNG: Ihr Programm sollte eine Möglichkeit besitzen, die Autostartmarke selbst zu löschen, oder das Betriebssystem mit der Einsprungadresse 0eh zu starten. Sonst können Sie die Autostartmarkierung nämlich nicht löschen, und Ihr Programm startet bis in alle Ewigkeit von selber! Sollte es dennoch passiert sein, ist ein kleiner Trick nötig: Legen Sie die /CE-Leitung Ihres NVRAM-Bausteins auf +5V und drücken Sie RESET. Damit ist das RAM nicht lesbar, und das Betriebssystem kann normal starten.

  - *Autostart löschen* löscht die Autostart-Markierung, so dass nach einem Reset des Zielsystems das CESY-Betriebssystem startet.

- *LAB-537* enthält ebenfalls weitere Unterpunkte, die die Arbeit mit dem LAB-537, seinem FLASH-Speicher und dem Bankswitch-System erleichtern:

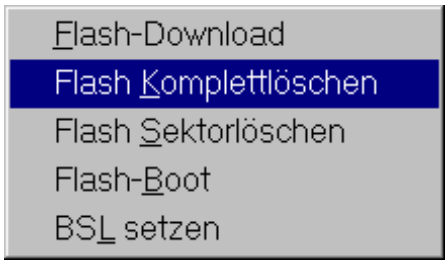


Abbildung 17: LAB537-Spezialbefehle

  - *Flash-Download* lädt einen Speicherbereich in den Flash-Speicher des LAB-537 herunter. Dabei werden nicht zugängliche Bereiche automatisch übersprungen und die Adressen korrigiert. Beachten Sie, dass das Flash ein EPROM ist, ein zweites Beschreiben also erst nach vorhergehendem Löschen möglich ist!

  - *Flash-Komplettlöschen* löscht das Flash komplett. Während des Vorgangs leuchtet die LED D3 am LAB-537 auf. Das Löschen dauert ca. 5-10 Sekunden, je nach Zustand des Speicherbausteins.

  - *Flash-Sektorlöschen* löscht einzelne 64k-Bereiche des Flash. Über ein Dialogfenster können Sie den zu löschenden Bereich wählen. Auch hier leuchtet die LED während des etwa 2 Sekunden dauernden Löschvorgangs. ACHTUNG – einige Flash-Bausteine erlauben kein Sektorlöschen!

  - *Flash-Boot* ist eine besonders interessante Eigenschaft. Durch das Schreiben einer Boot-Markierung wird bei RESET ein zuvor ins Flash heruntergeladenes Programm nach dem Starten in das RAM kopiert und das RAM danach als EPROM-Emulator konfiguriert (siehe Handbuch des LAB-537 und die Online-Hilfe). Schließlich wird das Programm ab Adresse 0000h gestartet. Sie müssen dazu die Länge Ihres Programmes eingeben. Die nicht zu-

gänglichen Bereiche am Ende jeder 16k-Seite des Flash werden automatisch berücksichtigt. Es gelten die selben Vorsichtsmaßnahmen wie beim Autostart beschrieben. Soll das LAB-537 Board wieder mit dem CESY-Betriebssystem starten, legen Sie einfach beim Start den Pin P6.7 (C24 an der Messerleiste) auf +5V. Damit kann die untere Flash-Bank nicht selektiert werden, und die Boot-Markierung wird nicht gefunden. Vergessen Sie nicht, die Boot-Markierung zu löschen! Ein erneutes Setzen ist nur nach dem Löschen dieses Flash-Sektors möglich.

Diese Funktion ermöglicht einen permanenten Einsatz des LAB-537 überall dort, wo das Einsetzen eines neuen EPROMs umständlich oder unmöglich wäre. Der Flash-Speicher ist sogar sicherer als ein normales UV-EPROM, weil er seine Information nicht durch UV- oder hochenergetische Strahlung verlieren kann.

- *BSL setzen*. Über ein Dialogfenster ist ein einfacher Zugriff auf das Bankswitch-Latch möglich. Sie können den aktuellen Stand des BSL auslesen und jedes Bit einzeln setzen. Dies funktioniert natürlich nur über das CESY-Betriebssystem, denn das BSL selbst ist nicht lesbar. Das Betriebssystem spiegelt das BSL in einer Speicherzelle des internen RAMs.

- *Direktausgabe:* Oftmals ist es praktisch, einen Speicherausschnitt direkt auf einer Druckerschnittstelle auszugeben. Viele EPROM-Emulatoren können so programmiert werden. Diese Aufgabe können Sie direkt von CESY aus erledigen. Sie müssen nur den benötigten Port (z. B. LPT1) eingeben. Achtung! Bitte keinen Doppelpunkt angeben. Die Funktion sollte auch mit den Seriellen Schnittstellen funktionieren, wobei die in den Systemeinstellungen getätigten Angaben für die Baudrate etc. hergenommen werden.

3.12 Das Terminal-Fenster

Über das Terminalfenster können Sie mit Ihrem Zielsystem frei kommunizieren. Insbesondere bei der Programmierung unter BASIC wird Ihnen diese Funktion eine große Hilfe sein. Es handelt sich dabei um ein einfaches ASCII-Terminal, das um die Cursorfunktionen im VT100-Standard erweitert wurde. Wenn Ihr Programm also mit dem CESY-Terminal klarkommt, wird es auch mit einem "echten" VT100 gut funktionieren.

Sie können im Optionen -- Terminal-Fenster die Schnittstelle frei konfigurieren. Achten Sie jedoch darauf, dass sich möglicherweise verschiedene Programmteile, wie z.B. Monitor, und das Terminalprogramm möglicherweise eine physikalische Schnittstelle teilen. Das kann zu Verwirrungen führen.

Neben den Konfigurationsknöpfen gibt es noch zwei weitere Checkboxen. Die Option "Echo on" bestimmt, ob jedes eingegebene Zeichen zusätzlich zur Schnittstelle auch noch auf dem Bildschirm ausgegeben werden soll. Meistens ist das nicht erforderlich. Dagegen sollten Sie die Option "CR->CR+LF" aktivieren, damit zusätzlich zum Carriage-Return auch noch eine Zeilenschaltung (Line Feed) ausgegeben wird.

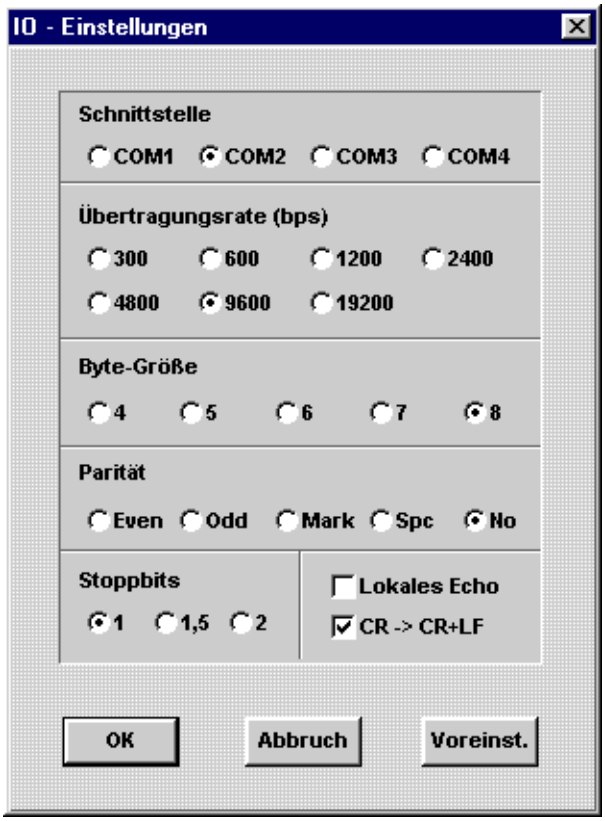


Abbildung 18: IO-Einstellungen

3.12.1 Troubleshooting

Wenn Ihre Terminaldarstellung ganz seltsam aussieht, kann das daran liegen, dass Ihr Zielsystem so schnell "feuert", dass das Terminalprogramm nicht damit fertigwerden kann. Bauen Sie dann kurze Verzögerungszeiten o.ä. ein. Dieses Problem kann übrigens auch unter BASIC auftreten. Ursache dieses Verhaltens ist, dass Windows nicht in der Lage ist, das Fenster so schnell zu scrollen, dass der Empfangspuffer nicht überläuft.

Achtung! Das Terminal verwaltet einen eigenen Satz von Einstellungen, der von den I/O-Einstellungen von Monitor/Download verschieden sein kann!

4 Hardware und CESY-Firmware

4.1 Hardware

CESY arbeitet mit jeder Hardware, die auf einem 8051-kompatiblen Mikrocontroller aufbaut. Die 80x51-Familie umfasst inzwischen Tausende von Typen, die sich in ihrer internen Hardware unterscheiden und demnach verschiedene SFR-Register besitzen. Sie sind jedoch alle Code-kompatibel, so dass Sie CESY für alle Typen benutzen können.

4.1.1 Download von Programmen

Bekanntlich ist der 80x51-Controller in Harvard-Architektur aufgebaut. Programme und Daten werden streng getrennt in separaten Speicherbänken aufbewahrt, die XRAM und Code-ROM genannte werden. Sollen nun Programme per Download zum Zielsystem geschickt werden, erweist sich diese Architektur als störend. Es muss ein RAM-Speicher her, der in beiden Speicherbänken sichtbar ist, so dass Programme aus dem RAM ausfürbar werden. Fast alle erhältlichen Controllerplatinen besitzen die Möglichkeit, einen solchen Speicherbereich einzurichten. Dazu werden das /RD und das /PSEN-Signal des Controllers AND-verknüpft (z. B. mit einem TTL-IC 7408) und an den /RD- bzw. /OE-Eingang des RAM-Chips gelegt. Falls Ihr System diese Möglichkeit nicht von sich aus bietet, kann sie mit ein paar Kabeln notfalls im fliegenden Aufbau schnell nachgerüstet werden.

4.1.2 Serielle Verbindung

Der PC und das Zielsystem müssen über ein serielles Kabel verbunden werden, damit CESY Zugriff auf das System hat. Ein solches Kabel kann bei beidseitig 9pol. Steckverbindern wie folgt aussehen:

- PC-Seite: Verbinden Sie die Pins 1+4+6 untereinander, sowie die Pins 7+8. Ab Windows 95 ist das nicht notwendig.
- Kabel: Verbinden der Pins 2,3 und 5 des Zielsystems mit den entsprechenden Pins des PCs. Ein Kreuzen der Pins 2 und 3 kann notwendig sein.

Beachten Sie, dass beide Schnittstellen mit den gleichen Übertragunsparametern 8N1 und der gleichen Übertragungsgeschwindigkeit laufen.

Beim ersten Start kann es vorkommen, dass die Übertragungsstrecke noch nicht synchronisiert ist. Sie sollten zunächst mit dem Monitor prüfen, ob die Übertragung reibungslos funktioniert, bevor Sie z. B. einen Download starten. Eventuell müssen Sie in der Windows-Systemsteuerung das FIFO des benutzten seriellen Kanals abschalten.

4.1.3 Vorbereitung für Einzelschrittbetrieb

Um Ihr System für den Einzelschrittbetrieb im Monitor klarzumachen, müssen Sie nur die INT0-Leitung (P3.2) Ihres Controllers auf Masse ziehen. Welcher Pin das

nun ist, schauen Sie bitte im Datenblatt des Controllers bzw. in der Anleitung Ihres Boards nach.

4.1.4 Einrichtung der Seriellen Schnittstelle

Die serielle Schnittstelle wird im Menü "Optionen -- IO-Parameter" eingerichtet. Sie können zwischen COM1 bis COM4 wählen. Baudraten von 300 bis 38400 bit/s sind möglich. Beachten Sie bitte, dass Sie die Schnittstelle nur dann einrichten können, wenn kein Programmteil darauf zugreift! Desweiteren ist zu beachten, dass die Windows-Unterstützung der Schnittstelle nicht besonders schnell ist. CESY benutzt aus Sicherheitsgründen die Original-Windows-Routinen. Bei einem P120-System ist oft schon 9600 bps das höchste aller Gefühle. Langsamere Systeme können unter Umständen noch niedrigere Baudraten erfordern.

4.1.5 Einrichtung des Terminals

Auch das Terminalmodul benutzt eine serielle Schnittstelle, die Sie im Menü "Optionen -- Terminal" konfigurieren können. Wenn das Terminal einen anderen Port die IO-Funktionen von CESY benutzt, gibt es keinerlei Probleme. Andernfalls sollte das Termianlprogramm die gleichen Übertragungsparameter und die gleiche Geschwindigkeit wie die IO-Funktionen benutzen, um Konflikte zu vermeiden.

4.1.6 Troubleshooting

Auch bei einem fertig gekauften Nullmodemkabel kann das Kreuzen der Leitungen 2 und 3 notwendig sein, nämlich genau dann, wenn das Zielsystem Modem-Pinbelegung hat. Außerdem gibt es Systeme, die ein besonderes Kabel erfordern, weil sie noch weitere Signale vom PC benötigen.

Beachten Sie, dass die Übertragungsgeschwindigkeit des Zielsystems von der Quarzfrequenz abhängt! Konsultieren Sie dafür das Kapitel "CESY-Firmware"!

Bei der Schnittstellenauswahl wird nicht überprüft, ob die Schnittstellen vorhanden und in der Windows-Installation richtig angemeldet sind.

4.2 Die CESY-Firmware

Damit CESY mit dem Zielsystem kommunizieren kann, muss auch darauf eine Software laufen. Diese Software liegt dem Programmpaket als Quelltext bei. Die Datei heißt "SYS51.SRC" für den 8031/8032, "SYS535.SRC" für den 80535 und „SYS537.SRC“ für den 80C537.

Dieses kleine "Betriebssystem" sorgt für Download, Upload, Programmstart, Monitorfunktionen und die Debuggerfunktionen. Da es über die serielle Schnittstelle kommuniziert, ist die Übertragungsgeschwindigkeit und die Quarzfrequenz im Systemprogramm festzulegen. Dies geschieht am Anfang des Quelltextes über die Label "BAUD" und "Q". Als Beispiel sind in den Programmen 9600 bit/s und 11,0592 MHz Quarzfrequenz/384 = 28800 eingetragen. Beim 80535/80537 kann auch mit dem Baudraten-Generator gearbeitet werden.

Des weiteren müssen noch die Einsprungadressen der Interrupts und der Autostart-Adresse eingetragen werden. Alle Sprünge auf die Interruptvektoren 0003h bis 006bh werden zu den entsprechenden Adressen in Ihrem Programmbereich weitergeleitet. Dazu ist das Label "D_Start" anzupassen. Es ist auf die Adresse 8000h voreingestellt. Ein Autostartprogramm würde so also an der Adresse 8000h starten, und alle Interrupts würden dorthin weitergeleitet. Dies ist besonders für das Funktionieren eines BASIC-Programmes wichtig, denn die Echtzeituhrfunktion benötigt den TIMER-Interrupt.

Wenn Sie also mit den voreingestellten Werten arbeiten können, müssen Sie nur noch das Programm assemblieren und können dann sofort loslegen!

4.2.1 Ressourcen

Von der CESY-Firmware verwendete Ressourcen:

- Data: 50h – 56h
- Stack: 30h – 4Fh (oder weniger, je nach Programm)
- Registerbank RB3 (18h – 1Fh) für das Betriebssystem
- Registerbank RB0 (00h – 17h) für Programme im Einzelschrittbetrieb
- Registerbank RB3 (18h – 1Fh) für Programme, die mit „Start“ gestartet werden.

Ihr Programm darf natürlich seine Registerbank frei auswählen, denn beim Rücksprung in CESY wird RB3 wieder hergestellt. Auch der Stackpointer darf neu initialisiert werden, aber ein Rücksprung zu CESY ist in diesem Fall nicht möglich (außer mit LJMP 0).

5 BASIC-Compiler

Bei weniger zeitkritischen Anwendungen ist es meist bequemer, wenn das Projekt in einer Hochsprache entwickelt werden kann. Die professionellste Hochsprache für Mikrocontroller ist sicherlich C, aber die Sprache C gilt nicht ganz zu unrecht als schwierig zu erlernen, und viele Anwender fühlen bevorzugen die weniger formelle Sprache BASIC.

Frühere CESY-Versionen (2.x) besaßen einen BASIC-Interpreter, mit dem sich zwar ordentlich programmieren ließ, der aber doch sehr langsam war. Der nun in CESY enthaltene Compiler übertrifft die Geschwindigkeit des Interpreters um das Hundertfache. Insbesondere die extrem schnelle 24bit-Integer-Arithmetik mit einem Zahlenumfang von ±8.388.607 macht den Compiler auch für komplexe Aufgabenstellung hinsichtlich Datenaufnahme und -analyse nutzbar.

Dennoch sollten Sie sich ein wenig auch in Assembler auskennen und mit den wichtigsten CESY-Funktionen sowie den Funktionen Ihres Controllerboards vertraut sein.

5.1 Übersetzen

Das BASIC-menü bietet zwei Befehle an:

• *Compilieren* startet den Compilerlauf gemäß den im „Optionen – BASIC“-Menü festgelegten Einstellungen und legt den Code direkt im virtuellen Speicher ab. Auf Wunsch können die Start- und Endadressen gleich ins Download-Fenster übernommen werden. Der Code ist fertig zum Ausführen.

• *Präcompilieren* startet den Compilerlauf gemäß Einstellungen, erzeugt aber keinen Opcode, sondern Assembler-Quelltext und stellt ihn in einem Editorfenster dar. Sie können, wenn es nötig ist, jetzt selbst Änderungen und Optimierungen am Code vornehmen.

5.2 Einstellungen

Sie müssen im Assembler-Menü den Prozessortypen einstellen, für den Sie Ihr Programm conpilieren möchten.

Der Compiler benötigt folgende Einstellungen, um ein funktionierendes Programm zu erstellen:

• *Startadresse*: Hier startet das Programm, dahinter folgt die Einsprungadresse für den Timer-Interrupt, der die Echtzeituhr bedient. Wenn hier "0000h" angegeben wird, ist das Programm zur Ausführung im ROM/EPROM geeignet und startet direkt nach einem Reset.

• *Data-Segment*: Variablen werden im internen Datenspeicher abgelegt. Da dieser jedoch recht beschränkt ist, passen in dem Datenspeicher eines 8031/8051 nur 15 Variablen. Bei Controllern mit 256 Byte Datenspeicher kann als Adresse "128" ein-

gegeben werden, dann ist Platz für alle 26 erlaubten Variablen. Außerdem ist dann mehr Speicher für den Prozessorstack vorhanden, was die Programmschachtelungs-Tiefe verbessert. Achtung: Sie müssen mindestens 5 Variablen im Data-Segment halten.

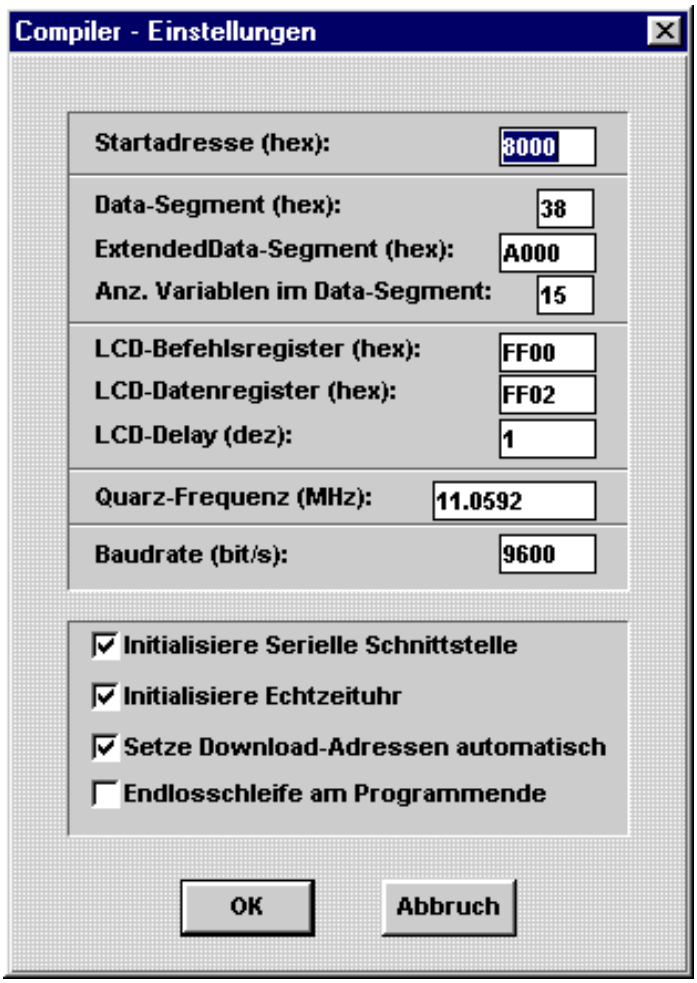


Abbildung 19: BASIC-Einstellungen

• *Extended Data-Segment*: Der Rest der Variablen, sowie das Array werden in der XRAM-Speicherbank gespeichert. Der Beginn dieses Segmentes muss festgelegt werden. Werden nicht mehr als Variablen benötigt, als in den internen Datenspeicher passen, so wird kein externer RAM-Speicher benötigt.

• *Anz. Variablen im Data-Segment*: siehe Data-Segment.

• *LCD-Register:* Wenn ein LCD-Display am Bus angeschlossen ist, kann es von BASIC aus angesteuert werden. Die Adressen dafür (nur für Schreibzugriff) können hier angegeben werden. Fast alle LCDs benötigen eine Verzögerungszeit nach einem Display-Kommando. Diese muss unter LCD-Delay angegeben werden. Es sind Zahlenwerte von 1 bis 15 erlaubt. Was für ein bestimmtes Display benötigt wird, muss ausprobiert werden. Einstellungen von 1 bis 4 sind für alle uns bekannten Displays ausreichend.

Der Compiler unterstützt auch Displays, die an einem I/O-Port angeschlossen sind. Dazu ist sowohl für die Befehls-, als auch für die Datenadresse die Portadresse anzugeben. Am gleichen Wert für beide Felder erkennt der Compiler, dass es sich um ein Port-Display handelt. Genaueres im Hardware-Kapitel.

• *Quarzfrequenz und Übertragungsgeschwindigkeit*: Die Quarzfrequenz muss für den genauen Gang der Echtzeituhr angegeben werden und ist außerdem für die serielle Schnittstelle wichtig. Die Übertragungsgeschwindigkeit für die serielle Schnittstelle muss korrekt angegeben werden, damit das BASIC über den PRINT-Befehl mit einem Terminal(-programm) kommunizieren kann.

Weiter muss noch angegeben werden, ob das Programm die *serielle Schnittstelle* oder die *Echtzeituhr* braucht. Falls nicht, können die Ressourcen anderweitig genutzt werden.

Als letztes geben Sie bitte an, ob nach Ausführung des Programms ein eine *Endlosschleife* gesprungen oder ob das *Betriebssystem gestartet* werden soll. Letzteres empfiehlt sich, wenn eine Autostart-Marke gesetzt wurde: dieser Einsprung fragt die Marke nicht ab, so dass die Möglichkeit besteht, eine solche Marke wieder zu löschen.

5.3 Aufbau eines BASIC-Programms

5.3.1 Zeilennummern und Sprungmarken:

Das CESY-Basic benötigt keine Zeilennummern. Zur Zuweisung von Sprungzielen müssen Sprungmarken (Labels) benutzt werden. Diese bestehen aus einem Ausrufezeichen und einer Zeichenkette, die mit einem Buchstaben beginnen muss und bis zu acht Zeichen lang sein darf.

Beispiel:

```
!Marke
 print "Hello, World!"
 goto Marke
```

5.3.2 Variablen:

Es gibt nur einen Variablentyp, der einen 24bit - Integer darstellt. Der Zahlenumfang reicht von -8.388.608 bis +8.388.607. Wenn Sie einen Fließkommawert darstellen wollen, beispielsweise 8,324 Volt, so rechnen Sie einfach mit Millivolt und kommen auf 8324 mV, was sich als Integervariable darstellen lässt.

Variablen werden durch Buchstaben dargestellt. Es gibt 26 Variablen, entsprechend der Buchstaben A bis Z, die gemäß der Einstellungen im Optionen-Fenster im internen oder externen Datenspeicher abgelegt werden. In der Voreinstellung werden die Variablen A bis P im internen Datenspeicher (DATA-Segment), und die Variablen Q bis Z im externen Datenspeicher (XRAM) abgelegt.

Beispiel

```
 a=8324
 b=a/1000
 print a," Volt"
 b=a-1000*b
 print b," Millivolt"
```

5.3.3 Konstanten

Oft wird ein Programm lesbarer, wenn numerische Konstanten durch sinnvolle Namen ersetzt werden. Dazu können zu Beginn des Programms numerische Konstanten deklariert werden. Übrigens sind alle Namen der SFR-Register zusätzlich vordefiniert. Diese vordefinierten Konstanten haben hohe Priorität, können also vom Be-

nutzer nicht verändert werden. Der Prozessortyp muss natürlich im Assembler-Menü richtig eingestellt worden sein.

Die mit **`#define`** definierten Konstanten sind case-sensitiv, müssen also in der korrekten Groß- und Kleinschreibung verwendet werden.

Beispiel:

```
#define OK_MELD = $c000
#define BITMUSTER = &10010110
outstr(OK_MELD)
out P1,BITMUSTER
```

Übrigens ist das SFR-Register "B" nicht zugänglich, da es vom Compiler mit der Variable "B" verwechselt würde.

5.3.4 Array

Zusätzlich zu den Variablen gibt es ein Datenfeld (Array). Es wird durch den Klammeraffen @ dargestellt. Dieses Datenfeld wird immer im XRAM-Segment abgelegt, und zwar hinter den 'normalen' Variablen. Die erlaubte Länge des Datenfeldes wird nicht überprüft, sie hängt allein vom Speicherplatz ab.

Beispiel

```
@(1)=2
@(2)=4
@(3)=8
print "2^3=",@(2)
```

5.3.5 Operatoren

Alle wichtigen Operatoren stehen zur Verfügung.

+	Addition	/	Division
-	Subtraktion	*	Multiplikation
^	Potenz	&	AND-Verknüpfung
|	OR-Verknüpfung		

Prioritäten: generell von links nach rechts; Punktrechnung vor Strichrechnung; Potenz wie Punktrechnung.

5.3.6 Verbinden von Befehlen

Mehrere Befehle in einer Zeile werden durch ein Semikolon verbunden.

Beispiel

```
a=13;print "Freitag, der ",a,"te"
```

5.3.7 Programm starten

Nach dem Compilieren und anschließendem Download kann das BASIC-Programm gestartet werden. Wenn Ihr Programm über die serielle Schnittstelle kommuniziert, sollten Sie vor dem Start-Befehl ein Terminalfenster öffnen, bzw. anklicken und es aktivieren. Geben Sie aber nichts ein, denn ein eingegebenes Zeichen würde das Monitorprogramm durcheinanderbringen. Starten Sie dann Ihr Programm. Der erste Befehl sollte ein delay(10) Befehl sein, der eine Sekunde wartet. Solange benötigt der Start-Befehl zum Umschalten in das Terminalfenster.

5.4 BASIC-Referenz

5.4.1 Ein-/Ausgabe-Befehle

• **PRINT** druckt einen airthmetischen Ausdruck oder einen konstanten Text aus. Arithmetische Ausdrücke werden ohne Formatierungsangaben rechtsbündig mit neun Stellen ausgegeben. Mehrere Ausdrücke oder Texte sind durch Kommata zu trennen. Mit dem Doppelkreuz wird der folgende Ausdruck auf n Stellen formatiert. Mit dem Prozent-Zeichen wird das Platzhalter-Zeichen festgelegt.

Beispiel	***Ausgabe***

`A=2`
`PRINT "123456789"`
`PRINT A,#4,A`
`PRINT #2,%'0',A`
`PRINT "TEST-",`
`PRINT "TEXT"`
	`123456789`
	`2  2`
	`02`
	`TEST-`
	`TEXT`

Die Ausgabe erfolgt zunächst über die serielle Schnittstelle. Zum Testen Ihrer Programme ist daher das in CESY eingebaute Terminal-Programm sehr nützlich. Mit Hilfe des DEVICE-Befehls kann die Ausgabe jedoch auf ein LCD-Display oder direkt in einen Speicherbereich umgeleitet werden.

- **INPUT *variable*** Einlesen numerischer Variablen vom Benutzer. Wird ein Text angegeben, so wird dieser gefolgt von einem Doppelpunkt ausgegeben. Danach wartet der Befehl auf eine Eingabe.

• *Beispiel*	• *Ausgabe*
`INPUT A` `INPUT "Text" B`	`_` `Text: _`

Die Ausgaben des Befehle erfolgen wie beim PRINT-Befehl. Die Eingaben werden ausschließlich von der seriellen Schnittstelle geholt. Daher macht dieser Befehl nur Sinn, wenn ein Terminal an das Zielsystem angeschlossen ist.

- **OUTCHAR *(wert)*** gibt ein ASCII-Zeichen aus. Auch dieser Befehl wird gemäß DEVICE-Anordnung auf der seriellen Schnittstelle, auf einem LCD-Display oder in einen Speicherbereich ausgegeben, wobei letzteres nicht viel Sinn macht.

Beispiel

```
OUTCHAR 65
```

- **INCHAR** holt ein Zeichen von der seriellen Schnittstelle.

Beispiel

```
A=INCHAR
PRINT A
```

- **OUTSTR *(adresse)*** gibt einen String, der ab der durch "adresse" bestimmten Stelle im Speicher steht, aus. Der String ist nullterminiert, das heißt erkann beliebig lang sein und wird durch eine binäre Null beendet. Um die Speicherverwaltung muss sich das BASIC-Programm kümmern. Ausführliche Erklärung und Beispiele siehe Kapitel String-Verarbeitung.
- **INSTR *(adresse)*** funktioniert ähnlich wie der INPUT-Befehl, nur dass anstatt eines numerischen Wertes ein String von der seriellen Schnittstelle geholt wird. Jedes eingegebene Zeichen wird geechot. Der String wird ab der durch "adresse" bestimmten Stelle in den Speicher geschrieben. Der String ist nullterminiert, das heißt erkann beliebig lang sein und wird durch eine binäre Null beendet. Um die Speicherverwaltung muss sich das BASIC-Programm kümmern. Ausführliche Erklärung und Beispiele siehe Kapitel String-Verarbeitung.

- **BYTE** ***(wert)*** gibt einen Wert als zweistellige Hexadezimalzahl auf die durch den DEVICE-Befehl spezifizierte Art aus.

Beispiel	***Ausgabe***
`BYTE(245)`	`F5`

- **WORD** ***(wert)*** gibt einen wert als vierstellige Hexadezimalzahl auf die durch den DEVICE-Befehl spezifizierte Art aus.

Beispiel	***Ausgabe***
`A=8000` `WORD(A)`	`1F40`

- **LWORD** ***(wert)*** gibt einen wert als sechsstellige Hexadezimalzahl auf die durch den DEVICE-Befehl spezifizierte Art aus.

Beispiel	***Ausgabe***
`A=-7654321` `LWORD(A)`	`8B344F`

- **TAB** ***(wert)*** gibt eine durch wert spezifizierte Anzahl von Leerstellen aus.

Beispiel	***Ausgabe***:

`PRINT "A"`
`TAB(4)`
`PRINT "B"`
	`A  B`

5.4.2 Struktur-Befehle

- **GOTO** ***Sprungmarke*** springt in die durch die Sprungmarke gegebene Zeile.

Beispiel:

```
!Marke
print"Hallo"
goto Marke
```

Dank der Strukturbefehle wie IF..THEN..ELSE..ENDIF oder REPEAT..UNTIL ist es nicht mehr notwendig, den GOTO-Befehl noch zu verwenden. **Guter Programmierstil zeigt sich in GOTO-freien Programmen!**

- **GOSUB *Sprungmarke* .. RETURN** ruft ein Unterprogramm auf und fährt nach dessen Bearbeitung mit dem Programm fort. Das Unterprogramm steht ab der durch die Sprungmarke bezeichneten Stelle und muss mit dem RETURN-Befehl abgeschlossen werden.

Beispiel	***Ausgabe***

`A=16`
`GOSUB Quadrat`
`PRINT #4,A`
`REM ... weiterer Text ...`
 	
`!Quadrat A=A*A`
`RETURN`
	`256`

- **FOR *laufvariable* = *anf* TO *end* (STEP *schritt*) .. NEXT *laufvariable***

Dies ist die Schleifenstruktur in BASIC. Der Block zwischen FOR und NEXT wird wiederholt und die Laufvariable dabei mit der durch die STEP-Anweisung definierten Schrittweite hochgezählt, und zwar bis die Laufvariable größer als end ist. Wenn die STEP-Anweisung weggelassen wird, beträgt die Schrittweite 1.

Beispiel	***Ausgabe:***

`FOR T = 1 TO 9`
`PRINT #2,T,`
`NEXT T`
	`1 2 3 4 5 6 7 8 9`

Die FOR-Schleife kann mit GOTO verlassen werden, auch wenn sie noch nicht bis zum Ende durchlaufen wurde. Das sollte man jedoch vermeiden, da es das Programm unübersichtlich macht.

- **IF (*Vergleich)* THEN *Programmblock* <ELSE *Programmblock*> ENDIF**

ACHTUNG! Dieser Befehl unterscheidet sich von V2.x! Zur Erhaltung der Kompatiblität gibt es den IFS-Befehl.

Bedingung. Nach IF folgt ein Vergleich. Es sind die Vergleichsoperatoren <, <=, =, >=, > und #(ungleich) möglich. Der Vergleichsausdruck sollte in Klammern stehen. Wenn der Vergleich positiv war, wird das Programm hinter "THEN" solange fortgeführt, bis es auf einen "ELSE"-Befehl trifft. Falls der Vergleich negativ war, wird das Programm ab dem folgenden "ELSE"-Befehl fortgeführt. Der ELSE-Block wird mit "ENDIF" abgeschlossen. Der ELSE-Block kann auch entfallen, dann wird der THEN-Block mit "ENDIF" abgeschlossen. Der "THEN"-Befehl kann weggelassen werden.

Beispiel	*Eingabe/ Ausgabe*		

!Start
INPUT A
IF (A=100) THEN
PRINT "A ist hun-
dert!"
ENDIF
	123_	250_	100_
			
			A ist hun-
			dert!

IF (A>200) THEN PRINT "Zu Gross"		Zu Gross	

ELSE
PRINT "Okay!"
ENDIF
	Okay!		Okay!

GOTO Start			

- **IFS** ***Vergleich Rest der Zeile***

ACHTUNG! Dieser Befehl dient zur Erhaltung der Kompatiblität zu CESY-BASIC V2.x. Benutzen Sie bei neuen Programmen bitte den erweiterten IF-Befehl.

Bedingung. Nach IFS folgt ein Vergleich. Es sind die Vergleichsoperatoren <, <=, =, >=, > und #(ungleich) möglich. Wenn der Vergleich positiv war, wird der Rest der Programmzeile ausgeführt. Falls der Vergleich negativ war, wird das Programm ab der folgenden Zeile fortgeführt.

Beispiel	*Eingabe/ Ausgabe*	
!Start INPUT A IFS (A=100) PRINT "Hundert!" PRINT A GOTO Start	123_ 123	100_ Hundert! 100

- **REPEAT** ***Programmblock*** **UNTIL** ***Vergleich***

Schleife mit Abbruchbedingung. Die Schleife startet mit REPEAT. Der bis zu UNTIL folgende Programmblock wird so lange wiederholt, bis der auf UNTIL folgende Vergleich positiv ist. Der Vergleich selbst ist äquivalent zu dem Vergleich bei IF.

Beispiel	*Eingabe/ Ausgabe*	
!Start INPUT A REPEAT A=A*2 PRINT "A=",A UNTIL A>=1000 GOTO Start	64_ A= 128 A= 256 A= 512 A= 1024	250_ A= 500 A= 1000

5.4.3 LC - Display

- **DINIT** initialisiert ein angeschlossenes LCD-Display und löscht die Anzeige. Die Befehls- und Datenadresse muss bei den Compiler-Einstellungen angegeben werden.
- **DEVICE** ***wert*** schaltet die Ausgabe aller Ausgabebefehle zwischen serieller Schnittstelle, LCD-Display oder XRAM-Speicher um.

  DEVICE 0 schaltet auf die serielle Schnittstelle,
  DEVICE 1 schaltet auf das LCD-Display,
  DEVICE v mit v>1 leitet die Ausgabe an den mit der Adresse v beginnenden Speicherbereich.

Beispiel:

```
DINIT
DEVICE 1
PRINT "HALLO LCD!"
DEVICE 0
PRINT "HALLO TERMINAL!"
DEVICE $A000
PRINT "HALLO SPEICHER!"
```

Die Möglichkeit, Ausgaben in den Speicher zu schreiben, ist für eine Stringverarbeitung sehr nützlich und wird dort näher beschrieben.

- **DISGO** ***position*** legt die Position der nächsten Ausgabe auf dem LCD-Display fest. Bei zweizeiligen Displays beginnt die zweite Zeile ab der Position 64.

Beispiel:

```
DINIT
DEVICE 1
DISGO 0
PRINT "Erste Zeile"
DISGO 64
PRINT "Zweite Zeile"
```

5.4.4

5.4.5 Speicherbefehle

- **POKE** ***adresse,byte*** schreibt ein Byte an die spezifizierte Adresse im XRAM-Segment.

***Beispiel*:**	***Ausgabe*:**
`POKE $A000,$80` `PRINT PEEK($A000)`	`128`

- **PEEK** ***(adresse)*** liest ein Byte aus dem XRAM-Segmant an der spezifizierten Adresse.

***Beispiel*:**	***Ausgabe*:**
`POKE $A000,$80` `PRINT PEEK($A000)`	`128`

- **PEEKC** ***(adresse)*** liest ein Byte aus dem Code-Segmant an der spezifizierten Adresse.

Beispiel	***Ausgabe***
`POKE $1000,$80` `PRINT PEEKC($1000)`	`3`

Da mit PEEK aus einer anderen Speicherbank gelesen wird, als mit POKE beschrieben werden kann, ergibt sich ein anderes Ergebnis.

- **POKED** ***adresse,byte*** schreibt ein Byte an die spezifizierte Adresse im DATA-Segment.

Beispiel	***Ausgabe***
`POKED $A0,$80` `PRINT PEEKD($A0)`	`128`

Beachten Sie bitte, dass das BASIC im DATA-Segment (in der Grundeinstellung) den gesamten Adressbereich von 0h bis 7fh benötigt. Sie sind also auf einen Controller mit mehr als 128 Byte internem RAM angewiesen, oder Sie müssen den Variablenspeicher (ab38h) benutzen. Siehe dazu auch CALL.´

- **PEEKD** ***(adresse)*** liest ein Byte aus dem DATA-Segmant an der spezifizierten Adresse.

• *Beispiel*	• *Ausgabe*
`POKED $A0,$80` `PRINT PEEKD($A0)`	`128`

- **CALL** ***adresse*** **r**uft ein Maschinenprogramm auf, das an der spezifizierten Adresse im Code-Segment steht. Das Programm muss den Stackpointer wiederherstellen und mit dem RET-Befehl enden.

Beispiel:

```
CALL $8146
```

Nachdem ein Maschinenprogramm aufgerufen wurde, muss es auf irgendeine Art und Weise mit dem BASIC-Programm wechselwirken können. Dazu kann, falls vorhanden, die XRAM-Bank dienen, die von BASIC aus mit PEEK und POKE zu lesen und zu schreiben ist. Falls jedoch kein XRAM vorhanden oder frei ist, muss

über den internen Datenspeicher kommuniziert werden. Dies kann per PEEKD und POKED geschehen. Alternativ bietet sich der elegante Weg der Variablen-Manipulation an. Dazu muss man wissen, dass die Variablen A..Z der Reihe nach ab der in den BASIC-Einstellungen gegebenen Adresse abgelegt werden. Die Voreinstellung dazu ist 38h. Das Format ist ein 24bit-Integer in der Reihenfolge Höchstwertiges Byte, mittelwertiges Byte, niedrigwertiges Byte (HML). So kann das Assemblerprogramm sich einen Satz von Variablen aussuchen, die es liest und verändert.

5.4.6 SFR-Register:

- **OUT adresse,byte** gibt einen Wert an ein SFR-Register aus.

Beispiel:

```
OUT 144,3
```

Setzt am Port 1 das erste und das zweite Bit.

- **IN *(adresse)*** liest einen Wert aus einem SFR-Register aus.

Beispiel:

```
PRINT IN(144)
```

Liest den Zustand von Port 1 ein.

- **SET *adresse,bit [,bit..]*** setzt ein oder mehrere Bits in einem SFR-Register. Nicht angegebene Bits werden nicht beeinflusst.

Beispiel	***Ausgabe***

`SET 144,0`
`SET 144,3,7`
`PRINT IN(144)`
	
	`137`

- **CLEAR *adresse [,bit [,bit...]] l***öscht ein oder mehrere Bits des angegebenen SFR-Registers, oder löscht es ganz, wenn keine Bits angegeben wurden. Wie beim SET-Befehl werden nur die spezifizierten Bits beeinflusst.

Beispiel:	**Ausgabe:**

`OUT144,255`
`CLEAR 144,1`
`CLEAR 144,3,5`
`PRINT IN(144)`
`CLEAR 144`
`PRINT IN(144)`
	
	`213`
	
	`0`

- **CHECK** ***(adresse,bit)*** fragt ein Bit an einem SFR-Register ab.

Beispiel	**Ausgabe**
SET 176,4 IF CHECK(176,4) PRINT "Ja." CLEAR 176,4 IF CHECK(176,4) PRINT "Ja."	Ja.

5.4.7 Serielle Schnittstelle:

- **SEROPEN** initialisiert die serielle Schnittstelle. Wenn im BASIC--Optionen-Fenster die Checkbox "Serielle Schnittstelle initialisieren" angeklickt wurde, wird die Schnittstelle automatisch zu Anfang des Programms initialisiert.
- **SERCLOSE s**chließt die serielle Schnittstelle, indem der damit verbundene Timer1 gestoppt wird. Dieser Timer steht nun für andere Aufgaben zur Verfügung.
- **SERBAUD (*hardwarerate*)** setzt eine neue Baudrate. Der Parameter ist nicht die Baudrate in bit/s, sondern berechnet sich aus der benötigten Baudrate und der Quarzfrequenz wie folgt:

hardwarerate = 256 - (Quarzfrequenz in Hz / (192* Baudrate in bit/s))

- **SERSTAT** liefert den momentanen Zustand der seriellen Schnittstelle. Wenn SERSTAT = 0 ist, so wurde kein Zeichen empfangen; ist SERSTAT = 1, so liegt ein Zeichen zum Abholen (z. B. mit INCHAR) bereit. Bitte beachten Sie, dass die Schnittstelle nicht gepuffert wird!

Beispiel:

```
A=256
IF SERSTAT A=INCHAR
PRINT A
```

Beachten Sie bitte einige Besonderheiten, die für 80x537er Prozessoren gelten (siehe unten).

5.4.8 Echtzeituhr

- **SECOND, MINUTE, HOUR, DAY, MONTH**

Diese Pseudovariablen enthalten, wenn die Software-Echtzeituhr aktiviert ist, die aktuelle Zeit sowie das aktuelle Datum. Sie können wie normale Variablen gesetzt und ausgelesen werden.

Beispiel:	***Ausgabe:***
`SECOND=0` `PRINT #2,%'0',HOUR,':',#2,MINUTE,':',#2,SECOND`	`17:10:00`

- **SETTIME** ***stunde,minute,sekunden*** setzt die Uhrzeit. Er wird eigentlich nicht benötigt, sondern dient zur Kompatiblität mit früheren BASIC-Versionen, in denen die Pseudovariablen nicht setzbar waren.

Beispiel:

```
SETTIME 23,50,20
```

- **SETDATE** ***tag,monat*** setzt das Datum. Er wird eigentlich nicht benötigt, sondern dient zur Kompatiblität mit früheren BASIC-Versionen, in denen die Pseudovariablen nicht setzbar waren.

Beispiel:

```
SETDATE 28,02
```

- **DELAY (*zehntelsekunden)*** hält das Programm für die spezifizierte Zeit, gegeben in Zehntelsekunden, an.

Beispiel:

```
DELAY(40)
```

wartet vier Sekunden.

- **OPENCLK** initialisiert die Echtzeituhr. Wenn im BASIC--Optionen-Fenster die Checkbox "Echtzeituhr initialisieren" angeklickt wurde, wird die Uhr automatisch zu Anfang des Programms initialisiert.
- **CLOSECLK** Stoppt die Echtzeituhr, indem der damit verbundene Timer0 gestoppt wird. Dieser Timer steht nun für andere Aufgaben zur Verfügung.

5.4.9 Analog-Digital-Wandler

- **GETAD(*MUX,RefLow,RefHigh*)**

Liest den ADC des 80C535/80C537/C552/C509 aus. Auch andere Controller, die einen ADC eigebaut haben, der auf dieselbe Weise angesprochen wird wie der 80C535, können benutzt werden.

Der erste Parameter selektiert im Multiplexer den gewünschten Eingang. Bei Benutzung eines 80C537 können vom BASIC alle 12 Eingänge benutzt werden.

Der zweite und dritte Parameter legen zusammen den Wertebereich fest, und zwar in Prozent der Referenzspannung Vref, die extern angelegt ist. RefLow und RefHigh dürfen sich um maximal 1V unterscheiden. Bei „echten“ 10-bit ADCs werden die Referenzspannungen nicht verwendet und müssen auf „0“ gesetzt werden.

Beispiel:	***Ausgabe:***
`A=GETAD(0,0,100)` `IFS A<64 THEN A=GETAD(0,0,25)`	
`PRINT A`	`129`

Die erste Zeile deckt den gesamten Wertebereich ab. Wenn der Wert im unteren Viertel lag, wird eine genauere Messung im Bereich 0..25% genommen.

5.4.10 Flash-Programmierung (nur mit LAB-537)

- **POKEF *adresse, byte***

Schreibt ein Byte an die spezifizierte Adresse im Flash-EPROM unseres LAB537.

- **PEEKF(*adresse*)**

Liest ein Byte von der spezifizierten Adresse im Flash-EPROM des LAB537.

Beispiel:	***Ausgabe:***
`POKEF $0734F0,$80` `PRINT PEEKF($0734F0)`	`128`

Beachten Sie bitte, dass dieser Befehl explizit auf den Flash-Speicher unseres LAB537 Mikrocontrollersystems abgestimmt ist. Da nach jedem 16kB Block ein Block von 256 Byte folgt, der nicht benutzt werden kann, erfolgt eine Adresskorrektur. Daher reicht der ansprechbare Bereich nur von 000000h bis 07DFFF.

Wie in jedem EPROM, können nur Bits von 1 auf 0 programmiert werden, nicht umgekehrt. Wird dies trotzdem versucht, kann der Flash-Baustein einen unkontrollierten Zustand annehmen. Testen Sie bitte vor Beschreiben, ob die Speicherzelle leer ($FF) ist.

- **ERASEF**

Löscht den gesamten Flash-Speicherbaustein

Beispiel:

```
ERASEF
DELAY(100)
```

Auch dieser Befehl ist explizit auf den Flash-Speicher unseres LAB537 Mikrocontrollersystems abgestimmt. Nach diesem Kommando muss 10 Sekunden gewartet werden, bis der Flash-Baustein wieder angesprochen wird. Diese Zeit benötigt das IC intern zum Löschen des Speichers. Der Befehl selbst wird in der üblichen Zeit ausgeführt (~50 µs). Für die Wartezeit trägt das BASIC-Programm Verantwortung.

5.4.11 Diverses

- **REM** kennzeichnet eine Zeile als Kommentar. Alles, was hinter dem REM-Befehl folgt, wird übersprungen.

Beispiel:	**Ausgabe:**

`REM PRINT "Kommentar"`
`PRINT "Kein Kommentar"`
 	
`PRINT "Mit Kommentar";`
`REM Kommentar`
	`Kein Kommentar`
	
	`Mit Kommentar`

- **LET**

Variablenzuweisung. Der Befehl LET wird zumeist weggelassen, und die Variablenzuweisung steht allein.

Beispiel:

```
LET A=2
B=A
```

- **ABS (*wert*)** stellt die mathematische Betragsfunktion dar.

Beispiel	***Ausgabe***

`A=-200`
`PRINT ABS(A)`
`PRINT ABS(200)`
	
	`200`
	`200`

- **RND(*grenze*)** liefert eine Pseudo-Zufallszahl zwischen 0 und grenze-1. Die erhaltenen Zahlen sind natürlich nicht wirklich zufällig, sondern werden über einen speziellen Algorithmus aus dem Programmcode generiert. Daher wiederholen sie sich irgendwann.

Beispiel:	**Ausgabe:**

`A=0`
`FOR I = 1 TO 1000`
`A=A+RND(10)`
`NEXT I`
`PRINT A`
	
	`4505`

- **HEX(*wert*), $*wert*** wandelt einen Hexadezimalwert in einen vom BASIC-Programm verarbeitbaren Integer-Wert um.

Beispiel	**Ausgabe**
`PRINT $A000` `PRINT HEX(A000)`	`40960` `40960`

- **BIN(*wert*), &*wert*** wandelt einen Binärwert in einen vom BASIC-Programm verarbeitbaren Integer-Wert um.

Beispiel	**Ausgabe**
`PRINT %111001110` `PRINT BIN(1000)`	`462` `8`

- ***Quotation*** Oftmals kommt es vor, dass etwa einer Variablen ein ASCII-Code zugewiesen werden muss. Dieses kann mit einem einfachen Anführungszeichen (‘) geschehen. Beim PRINT-Befehl ist diese Funktion außer Kraft gesetzt.

Beispiel	**Ausgabe**

`a='A'`
`print a`
`print 'a'`
	
	`65`
	`a`

- **LENGTH (*adresse*) e**rmittelt die Länge des Strings, der ab der angegebenen Adresse im XRAM-Speicher steht. Beachten Sie bitte, dass zusätzlich zur Länge der Nutzinformation auch noch die Speicherstelle, die die terminierende Null in An-

spruch nimmt, berücksichtigt werden muss. Im Abschnitt Stringverarbeitung der Online-Hilfe wird diese Funktion ausführlicher erläutert.

Beispiel	***Eingabe/ Ausgabe***

`PRINT "Eingabe: ",`
`INSTR ($a000)`
`PRINT`
`"Länge:",#3,LENGTH($a000)`
	`Eingabe: Test_`
	
	`Länge:  4`

- **EQSTR (*adresse1, adresse2*)** vergleicht die Strings, die an den gegebenen Adressen im XRAM-Speicher stehen. Die Funktion gibt 1 aus, wenn beide Strings gleich lang sind und aus den gleichen Zeichen bestehen. Weiteres im Kapitel "Stringverarbeitung".

Beispiel	***Eingabe/ Ausgabe***

`PRINT "Eingabe 1: ",`
`INSTR($a000)`
`PRINT "Eingabe 2: ",`
`INSTR($a010)`
`IF EQSTR($a000,$a010)`
`PRINT "Gleich!"`
	`Eingabe 1: CESY_`
	
	`Eingabe 2: CESY_`
	
	`Gleich!`

- **INCLUDE("filename", Symbol)** dient zum Einbinden von Assembler-Routinen in BASIC-Programme. Er sollte nur von erfahrenen Benutzern verwendet werden. Das Assemblerprogramm mit dem angegebenen Dateinamen ("filename") wird beim Compilieren hinter dem Programm angehängt. Es erhält den angegebenen Symbolnamen, unter dem es für den BASIC-Befehl "RUN",der zum Starten von einzelnen Routinen dient, bekannt ist.

Das Assemblerprogramm muss im Quelltext vorliegen und sich mit CESY problemlos assemblieren lassen. Weitere Include-Dateien dürfen eingebunden werden. Labels dürfen frei verwendet werden; sie sind dann in der Include-Datei lokal, d. h. es dürfen keine Programmteile aufgerufen werden, die nicht in der Include-Datei vorkommen. Makros sind erlaubt.

Es sind zwei Arten von Include-Dateien erlaubt:

1. Dateien mit nur einem Programm (Einsprung),
2. Dateien mit mehr als einem Programm

Im ersten Fall muss das Programm gleich in der ersten Zeile beginnen und mit dem RET-Befehl abgeschlossen werden. Im zweiten Fall muss am Anfang ein Sprungtabelle stehen, die folgendes Format aufweist:

```
LJMP  sub1
LJMP  sub2
LJMP  sub3

  ...
```

Jeder Programmteil ist mit RET abzuschliessen.

- **RUN(Ausdruck, Symbol<:ID>)** startet ein Assembler-Unterprogramm, das in einer mit dem INCLUDE-Befehl eingebundenen Datei vorhanden ist. Als Parameter kann ein arithmetischer Ausdruck oder eine Variable angegeben werden. Dieser wird dem Assemblerprogramm in den Registern R5, R6 und R7 (LMH-Byte) übergeben. Welche Include-Datei gemeint ist, wird aus dem angegeben Symbol ersichtlich. Enthält die Include-Datei mehrere Unterprogramme und eine Sprungtabelle, so muss über die ID nach dem Doppelpunkt das entsprechende Unterprogramm ausgewählt werden.

Beispiel:

```
INCLUDE("INCBAS1.INC",Switch)
INCLUDE("INCBAS2.INC",Stop)
RUN(3,Switch:0)
RUN(3,Switch:1)
RUN(0,Stop)
```

Zunächst werden zwei Dateien eingebunden. Die erste enthält zwei Unterprogramme, die über den Identifier selektiert werden. Die zweite Include-Datei besitzt nur einen Befehl, der keiner weiteren Spezifizierung bedarf.

Ein ausführlicheres Beispiel findet sich unter den Dateinamen "INCLUDE.BAS", "INCBAS1.INC" und "INCBAS2.INC" auf der CESY-Diskette/-CD.

- **STOP** beendet ein BASIC-Programm, indem ein Sprung auf das Programmende vorgenommen wird. Je nach Einstellung springt der Prozessor in eine Endlosschleife oder zum Monitorprogramm.

5.4.12 Stringverarbeitung

Das CESY-Basic bietet eine zugegebenermaßen spartanische, aber dennoch sehr effektive Stringverarbeitung an. Diese beruht auf dem System der nullterminierten Strings, das der Sprache C entlehnt wurde. Dabei wird ein String einfach im Speicher abgelegt, und hinter den String wird eine binäre Null geschrieben. Dadurch verlängert sich die effektive Länge des Strings um ein Byte. Um die Verwaltung des Speicherbereiches für die Strings muss sich das Programm kümmern.

Da Mikrocontroller üblicherweise für Steuerungsaufgaben verwendet werden, dienen Strings vor allem zur Benutzerführung. Dazu sind die vorhandenen Stringbefehle voll ausreichend. Mit ein paar Kniffen ist aber noch mehr möglich, so können Benutzereingaben verwaltet und manipuliert werden. Eine Anwendung für so etwas wäre z. B. ein in ein Telefon integriertes Telefonbuch.

- Alphanumerische Eingaben

  Mit dem Befehl INSTR kann über die serielle Schnittstelle eine Zeichenkette vom Benutzer angefordert werden, die dann direkt in den Speicher geschrieben wird.

- Numerische Eingaben und Programmergebnisse

  können mit Hilfe des DEVICE-Befehls an die gewünschte Speicherstelle geschrieben werden. Dabei wird mit DEVICE die Speicherstelle festgelegt, an der die Zeichenkette beginnen soll.

- Ausgaben aus dem Speicher

  sind mit dem OUTSTR-Befehl möglich. Wenn mit dem DEVICE-Befehl dann noch eine Zieladresse angegeben wird, hat man einen einfachen und schnellen Kopiervorgang.

- Die Länge eines Strings

  kann mit dem LENGTH-Befehl leicht ermittelt werden.

- Vergleiche:

  Zwei Strings werden mit Hilfe der EQSTR-Funktion verglichen.

Beispiel 1	***Eingabe/ Ausgabe***	
A=$a000 PRINT "Ihr Name: ",	Ihr Name: Wurst_	Ihr Name: Thomas_
INSTR(A) B=A+LENGTH(A)+1 PRINT "Ihr Vorname: ",	Ihr Vorname: Hans_	Ihr Vorname: Thomas_
INSTR(B) C=B+LENGTH(B)+1 DEVICE C OUTSTR(B) DEVICE C+LENGTH(C)+1 OUTSTR(A) POKE C+LENGTH(c),' ' DEVICE 0 PRINT "-> ",	-> Hans Wurst	-> Thomas Thomas
OUTSTR(C) IF EQSTR(A,B) PRINT "Vor- und Nachname sind ja gleich!" ENDIF		Vor- und Nachname sind ja gleich!"

Erklärung des Beispiels: zunächst wird die Adresse festgelegt, ab der Strings gespeichert werden sollen. Achten Sie darauf, dass es keinen Konflikt mit den Adressen für Variablen und Arrays gibt. Danach wird über ein Terminal der Name des Benutzers eingelesen. Mit Hilfe dessen Länge ist dann die Adresse des nächsten Strings zu bestimmen. Dort wird der Vorname abgelegt. Wieder wird die Adresse des nächsten Strings bestimmt. Jetzt soll der gesamte Name in richtiger Reihenfolge in den Speicher geschrieben werden. Der DEVICE-Befehl legt die Zieladresse C fest, mit OUTSTR wird zuerst der Vorname, der ab der Adresse B steht, dann dahinter der Nachname geschrieben. Zwischen diesen beiden Strings steht noch die Terminierung, die zu guter Letzt mit einem Leerzeichen überschrieben wird. Dadurch sind nun die Strings verbunden!

Beispiel 2	**Ausgabe**			

`Print "Eintrag Nr. ", #1,i`
`Print "Name: ",`
 	
`Instr(A)`
`@(i)=A`
`A=A+LENGTH(A)+1`
`Print "Telefon: ",`
 	
`Instr(A)`
`A=A+LENGTH(A)+1`
`Next i`
`Print`
`print "Suche Namen: ",`
 	
`instr($a000)`
`for i = 0 to 4`
`if EQSTR($a000,@(i))`
`print "Telefon: ",`
`;OUTSTR(@(i)`
`+LENGTH(@(i))+1)`
`endif`
`next i`
	`Eintrag Nr. 0`
	`Name:`
	`Angela_`
	
	`Telefon:`
	`23756_`
		`Eintrag Nr. 1`
		`Name:`
		`Gerhard_`
		
		`Telefon:`
		`110573_`
			`Eintrag Nr. 2`
			`Name:`
			`Hans_`
			
			`Telefon:`
			`856739_`
				`Suche Namen:`
				`Gerhard_`
				
				`Telefon:`
				`110573`

Dieses Beispiel demonstriert, wie leicht eine Suchfunktion implementiert werden kann. Die Namen und Telefonnummern werden direkt hintereinander in den Speicher geschrieben. Die Adresse eines jeden Datenblocks vermerkt das Programm in einem Array. Bei der Suche werden in einer FOR..NEXT-Schleife alle Einträge durchgegangen und mit dem Suchbegriff verglichen. Die gefundene Nummer wird schließlich ausgegeben.

5.5 Interrupts in BASIC

In dieser CESY-Version ist eine Erweiterung enthalten, die das Einbinden von Interrupt-Routinen in BASIC-Programme ermöglicht. Hierzu einige Hinweise:

Eingebunden können nur Assembler-Programme werden. Sie müssen eine besondere Form haben. Siehe dazu die Datei INT_DEMO.INC. Die Form ist ähnlich der für normale Include-Dateien; es müssen aber bestimmte Label-Namen verwendet werden.

- Das Einbinden geschieht mit dem Include-Befehl "INCLUDE(symbol,"filename",i)". Das Besondere daran ist das ",i", welches an die normale Syntax angehängt wird. Es sorgt dafür, dass eine Interrupt-Sprungtabelle in die Laufzeitbibliothek aufgenommen wird, die auf die eingebundene Include-Datei verweist.
- Das Setzen von Bits in den Interrupt-Konfigurationsregistern muss vom Basic-Programm aus "von Hand", also mittels OUT oder SET etc. durchgeführt werden.
- Wichtig! Sie als Programmierer müssen darauf achten, dass alle Register richtig auf den Stack gerettet werden und nach dem Abarbeiten des Programms auch wieder restauriert werden!
- Es müssen ALLE in der Vorlage "INT_DEMO.INC" angegebenen Einträge in die Sprungtabelle verwendet werden.

5.6 Besonderheiten bei C517/C509 Prozessoren

Der verbreitete SAB-C517 (bzw. 80C537) von Infineon und seine Ableger (z.B. SAB-C509) verfügen über einige Besonderheiten, die vom BASIC unterstützt werden. Insbesondere sind dies der Arithmetik-Koprozessor und der flexible Baudraten-Generator. Der 537-Modus wird eingeschaltet, indem im Assembler-Menü die entsprechende Option (80x537,C517A,C509) aktiviert wird.

5.6.1 Baudraten-Generator

Der Baudraten-Generator wird standardmäßig benutzt. Damit erübrigt sich der Befehl "SERCLOSE". Die Baudrate für den SERBAUD-Befehl errechnet sich wie folgt:

hardwarerate = 256 - (Quarzfrequenz in Hz / (32 Baudrate in bit/s))*

Man rechnet nach, dass für einen 11.0592 MHz-Quarz und 9600 bps der Wert von 220 nötig ist. Der Timer 1, der sonst für die serielle Schnittstelle benutzt wird, ist frei.

5.6.2 Arithmetik-Unit (MDU)

Dieser arithmetische Coprozessor wird für Multiplikation und Division benutzt. Für Multiplikation bringt dies aufgrund des sowieso schon schnellen Multiplikations-Algorithmus nicht viel. Bei Division dagegen beträgt der Geschwindigkeitsvorteil etwa einen Faktor von 15. Beachten Sie bitte, dass die MDU nicht benutzt werden kann, wenn der Divisor größer ist als 65535. In diesem Falle wird auf den Standard-Algorithmus zurückgegriffen.

5.7 Fehlermeldungen

Der Compiler meldet jeweils den ersten Fehler, den er findet und bricht danach den Compiliervorgang ab. Folgende Fehlermeldungen können auftreten.

- **Undefined Label Error: e**in GOTO- oder GOSUB-Befehl hat ein Sprungziel angegeben, das nicht definiert wurde.
- **Syntax Error: d**iesen Befehl gibt es nicht.
- **String constant exceeds line Error: e**ine Stringkonstante bei einen PRINT-Befehl darf maximal 64 Zeichen lang sein und muss auf beiden Seiten mit einem Anführungszeichen " abgeschlossen werden.
- **Hex constant too long Error: e**ine hexadezimal angegebene Konstante darf aus maximal 6 Zeichen bestehen.
- **Illegal expression Error: e**in arithmetischer Ausdruck ist in dieser Form nicht zulässig.
- **Missing bracket Error: e**ine Klammer fehlt.
- **Input without reference Error: e**in INPUT-Befehl weiß nicht, in welche Variable das Ergebnis geschrieben werden soll.
- **Illegal label definition Error: e**in Label darf so nicht definiert werden.
- **Illegal Character in label Error: e**in Label darf nur aus den Zeichen 'A'..'Z','0'..'9','_' bestehen.
- **Label defined twice Error: e**in Label darf nur einmal definiert werden, um eine eindeutige Zuweisung zu ermöglichen.
- **NEXT without FOR Error: d**er Compiler weiß nicht, was er mit dem einsamen NEXT tun soll.
- **NEXT without variable Error: d**er NEXT-Befehl ist nur mit einer Bezugsvariablen gültig.

- **Too many nested IFs Error: m**aximal 20 IF-Anweisungen können verschachtelt werden.
- **Too many nested REPEATs Error: m**aximal 20 REPEAT-Anweisungen können verschachtelt werden.
- **IF without ENDIF Error: z**u einer IF-Anweisung gehört immer ein ENDIF, das die Schachtelung schließt. Ausweg: benutzen Sie „IFS“.
- **REPEAT without UNTIL Error: z**u einer REPEAT-Anweisung gehört immer auch ein UNTIL, das die Schachtelung schließt

5.7.1 Laufzeitfehler:

Die meisten Befehle werden zur Laufzeit ohne jede Prüfung ausgeführt. Dies erfordert eine große Sorgfalt beim Programmieren. Falls fatale Fehler zur Laufzeit auftreten, die den Ablauf des Programmes gefährden würden, wird eine Fehlermeldung über die serielle Schnittstelle ausgegeben, sofern sie geöffnet ist. Ansonsten bleibt das Programm an der Stelle stehen, bis ein RESET ausgelöst wird. Die einzigen fatalen Laufzeitfehler sind Arithmetikfehler: Überlauf oder Division durch null.

6 Der C-Compiler

CESY ist ab Version 8 mit einem C-Compiler ausgestattet. Der Compiler ist jedoch nicht wie der BASIC-Compiler ein echter Teil von CESY, sondern es handelt sich um ein großes Open-Source Projekt, das im Internet von vielen gemeinsam arbeitenden Informatikern entwickelt wurde und kontinuierlich weiterentwickelt wird. Der Name SDCC steht für „Small Devices C Compiler“, auf Deutsch „C-Compiler für kleine Prozessoren“. Zufälligerweise sind die Initialen des Hauptautors ebenfalls „S.D.“.

CESY bindet den SDCC-Compiler ein, so dass seine Benutzung erheblich vereinfacht wird. Der Editor enthält einen Satz von Textfarben für die C-Syntax. Zum Compilieren wird der Text im Editor automatisch gespeichert und dann ein externer Programmaufruf zum C-Compiler gestartet.

CESY ruft dann SDCC über eine DOS-Kommandozeile auf, die eine Vielzahl von Kommandos enthalten kann. Um die Erstellung dieser Kommandozeile und ihren Aufruf kümmert sich CESY. Der übersetzte Code wird direkt wieder geladen und kann mit dem Monitor bearbeitet werden. Außerdem entsteht ein Log-File, das im Falle von Fehlern oder Warnungen ebenfalls angezeigt wird. Über das Log-File können dann die Fehlerstellen im Editor komfortabel angesprungen werden.

6.1 Stand der Entwicklung

In der vorliegenden Version können Programme, die aus einer einzigen Datei bestehen, mit CESY übersetzt werden. Für größere Projekte gibt es ein einfach zu bedienendes Projektmanagement, über das Projekte aus mehreren Dateien verwaltet übersetzt und gelinkt werden können.

SDCC kann auch zum Übersetzen von Bibliotheken verwendet werden. Dies erfordert jedoch die Benutzung von der Kommandozeile. Hierzu ist eine CESY-Anpassung in Arbeit. Über Kommandozeilen erzeugte Bibliotheken können natürlich auch mit CESY verwendet werden.

6.2 Open Source, Urheberrecht und Garantieverzicht

SDCC ist unter der GNU Public License veröffentlicht. Das heißt, dass der Compiler (aber nicht CESY!) frei im Internet verfügbar ist, und zwar mitsamt den Quelltexten. Jeder darf das Programm benutzen und kostenlos weiterverbreiten (gilt auch nicht für CESY!).

Dies impliziert aber auch, dass von unserer Seite keine Garantie auf das Funktionieren des SDCC gegeben werden kann (obwohl es besser funktioniert als so manche teurere Software). Vor allen kann nicht garantiert werden, ob SDCC für den von Ihnen gewünschten Zweck verwendbar ist.

6.3 Speicherbereiche

Im Menü „SDCC-Optionen“ können die Speicherbereiche, die das C-Programm im Zielsystem einnimmt, eingestellt werden. Für den Anfang reicht es aus, wenn Code-Segment und XRAM-Segment auf Ihr Zielsystem angepasst werden. Die Einstellmöglichkeiten im Einzelnen:

- Code segment: hier startet Ihr Programm. Achten Sie darauf, dass die Interrupt-Adressen richtig angesprungen werden. Für unser LAB-537 wäre die Einstellung „0000“, für die INDU-Boards „8000h“. Letzteres passt auch für die meisten Boards anderer Hersteller. Soll das Programm schließlich in ein EPROM gebrannt werden, so muss die Code-Adresse natürlich auf „0000“ gesetzt werden.
- XRAM segment: hier legt das Programm Variablen und Strings ab. Kleinere Programme werden in der Regel ohne XRAM auskommen.
- Data-Segment: diese Einstellung sollten Sie möglichst auf „Auto“ lassen. Damit wird das Data-Segment, also der interne Datenspeicher, automatisch verwaltet, und zwar nach folgender Regel:
    - Variablen werden ab Adresse 08h abgelegt.
    - Wenn weitere Registerbänke verwendet werden, z.B. im Rahmen von Interrupts, verschiebt sich die Anfangsadresse entsprechend.

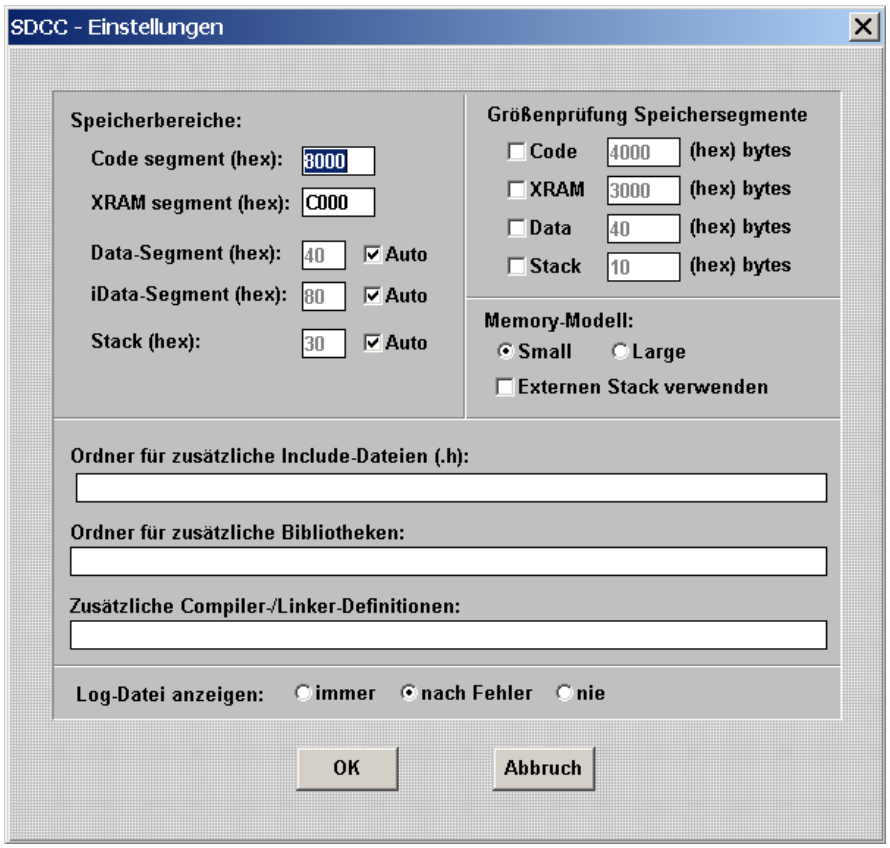


Abbildung 20: SDCC-Optionen

  - Wenn Bit-Variablen verwendet werden, so werden diese im bit-addressierbaren Speicher von 20h bis 27h abgelegt. Variablen werden dann dahinter abgelegt.

- iData-Segment: bezeichnet den internen Datenspeicher, der nur indirekt adressierbar ist. Dieser liegt eigentlich immer bei 80h. Eine eigene Einstellung macht nur Sinn, wenn aus irgendwelchen Gründen ein Bereich freigehalten werden soll.
- Stack: der Stack wird in der automatischen Einstellung hinter den Variablen in das Daten-Segment gelegt.

Ferner kann noch eine Größenprüfung der einzelnen Segmente eingeschaltet werden.

Wichtig ist das Speichermodell. Im Modus „small“ werden alle deklarierten Variablen im internen Datenbereich abgelegt, sofern dies nicht in der Deklaration anders bestimmt wird. Dies ermöglicht einen sehr effizienten Code. Wird der Modus „large“ verwendet, so werden die Daten im externen RAM abgelegt, was natürlich umständlicher ist (für Assembler-Kenner: MOV DPTR,#xxx, MOVX A,@DPTR, MOV Rx,A = 5 Bytes; MOV Rx,xx = 2 Bytes!) und zu langsameren Programmen führt.

Bei zusammengesetzten Programmen müssen immer alle Bibliotheken und Programmteile im gleichen Speichermodell übersetzt werden! Die Standardbibliotheken sind bereits für beide Modelle übersetzt (Ordner lib\small oder lib\large).

Die Option „Externen Stack verwenden“ wird von der derzeitigen Version des SDCC-Compilers nicht sauber unterstützt und ist nur für Experimente gedacht.

Schließlich können noch Suchpfade für Include-Dateien und Bibliotheken sowie zusätzliche Compiler-Linker-Definitionen übergeben werden.

CESY kann nach dem Compilieren eine LOG-Datei anzeigen. Dies ist sicherlich sinnvoll, wenn der Compiler einen Fehler gemeldet hat. Man kann auch nach jedem Übersetzten die Log-Datei anzeigen lassen, dann kann bei der Programmerstellung gleich auch auf Warnungen geachtet werden.

6.4 Compilieren

Das Übersetzen eines C-Programms ist genauso einfach wie das eines Assembler- oder BASIC-Programms. In der Hauptmenüzeile findet man das C-Compiler-Menü unter „C“. Hier gibt es folgende Unterpunkte:

Abbildung 21: Compiler-Menüs

- *Compilieren*. Das Programm, das sich im Editor befindet, wird übersetzt und gelinkt. Nach dem Übersetzen wird im Erfolgsfall der erzeugte Code in den virtuellen Speicher geladen. Je nachdem, wie CESY konfiguriert ist („Optio-

nen – Eigenschaften“), wird eine .BIN und eine .HEX-Datei erzeugt. Im Falle eines Fehlers wird die vom Compiler erzeugte LOG-Datei angezeigt. Klicken auf eine Fehlermeldung springt in das Editorfenster auf die entsprechende Zeile.

- *Compilieren ohne Linker*. Hier findet kein Link-Vorgang statt, und es werden keine binären Daten erzeugt. Dieser Befehl dient zum Testen von Unterprogrammen, die in ein Projekt eingebunden werden und nur im Zusammenhang mit einem Hauptprogramm (das eine main() Routine enthält) funktionsfähig ist.
- *Comp/Down/Start*. Übersetzt und linkt das Programm. Wenn erfolgreich, wird das Programm zum Zielsystem geschickt und dort gestartet. Hierzu muss in den CESY-Eigenschaften die Option „Setze Download-Adressen automatisch“ aktiviert sein.
- *Projekt*. Befehle für den Projekt-Manager.

! **Der Linker gibt Meldungen, dass er aufgerufene Funktionen nicht gefunden hat, lediglich als Warnung aus. Das Programm wird trotzdem erzeugt und in den virtuellen Speicher geladen! Im Falle mysteriöser Programmfehler schauen Sie bitte in die Log-Datei!**

6.5 Projekt-Manager

Ein größeres Projekt besteht üblicherweise aus folgenden Komponenten:

- Hauptprogramm, d.h. ein Programm, das einen main()-Block enthält.
- Unterprogramme, d.h. Programme, die keinen main()-Block enthalten und in das Hauptprogramm eingebunden werden
- Zu jedem Unterprogramm eine entsprechende Header-Datei (.h), die die Prototypen der im Unterprogramm bereitgestellten Funktionen enthält.

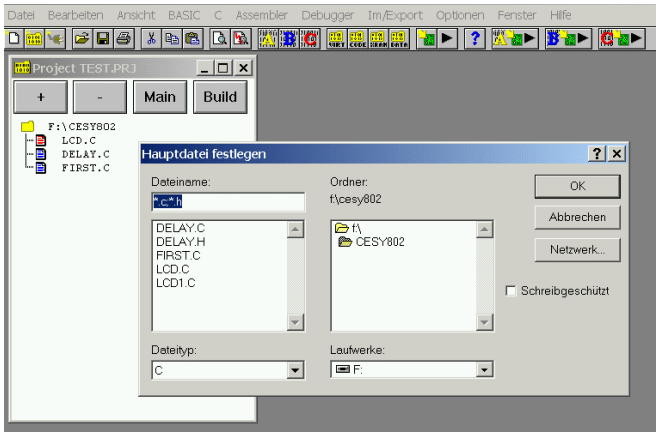


Abbildung 22: Projekt-Manager beim Festlegen der Hauptdatei.

Der Projekt-Manager dient zur Definition der Dateien, d.h. welche Datei als Hauptprogramm und welche anderen Dateien als Unterprogramme dienen. Header-Dateien werden in das Projekt nicht durch den Projekt-Manager, sondern über #include““-Anweisungen im Hauptprogramm eingebunden.

6.5.1 Öffnen / Neues Projekt

Mit dem Öffnen-Befehl wird ein bestehendes Projekt geöffnet oder ein neues Projekt erstellt. Für ein neues Projekt muss man den Dateinamen angeben. CESY fragt dann, ob unter diesem Namen ein neues Projekt erstellt werden soll.

Die Projekt-Datei (*.PRJ) und alle Programmdateien müssen im selben Ordner stehen!!!

Nach dem Öffnen geht das Projektfenster auf, in dem der Projektordner angezeigt wird. Den Projektordner kann man nur durch das Festlegen der Hauptdatei ändern!

6.5.2 Hauptdatei festlegen

Der Befehl zum Festlegen der Hauptdatei ist über das C/Projekt-Menü oder direkt über den „MAIN“-Knopf im Projektmanager-Fenster erreichbar. Dies sollte der erste Schritt in jedem Projekt sein.

Übrigens muss, wenn man mit dem Projektmanager arbeitet, nicht unbedingt jede Datei in einem Editorfenster geöffnet sein. Öffnen Sie nur die Dateien, für die das sinnvoll ist, meist die Hauptdatei.

6.5.3 Dateien zum Projekt hinzufügen oder entfernen

Diese Befehle sind über das C/Projekt-Fenster oder über die Knöpfe „+“ und „-“ aus dem Projektmanager-Fenster erreichbar. Zum Entfernen muss die entsprechende Datei im Projektmanager-Fenster angeklickt werden. Die Datei wird natürlich nicht auf dem Datenträger gelöscht, sondern nur aus der Projektliste entfernt.

6.5.4 Projekt übersetzen

Der Übersetzungsprozess für ein Projekt aus mehreren Dateien wird traditionell „Build“ (sprich: „Bild“) genannt. Dieser Begriff soll auch in der deutschen CESY-Version verwendet werden.

Gestartet wird die Projektübersetzung wieder über das Popup-Menü oder über den „Build“-Knopf. Hier gibt es einige Besonderheiten:

- Nur die LOG-Datei des Hauptprogramms wird angezeigt. Die LOG-Dateien der Unterprogramme werden zwar erzeugt, aber nicht geladen.
- Unterproramme sollten also vorher auf offensichtliche Übersetzerfehler getestet werden. Dazu dient der Befehl „Compilieren ohne Linker“.
- Bei erfolgreicher Übersetzung wird der Binärcode des gesamten Programms in den virtuellen Speicher geladen und kann dort weiterverarbeitet werden.
- Header-Dateien müssen mit #include“unter.h“ und nicht mit #include<unter.h“ eingebunden werden.

Der Linker gibt Meldungen, dass er aufgerufene Funktionen nicht gefunden hat, lediglich als Warnung aus. Das Programm wird trotzdem erzeugt und in den virtuellen Speicher geladen! Im Falle mysteriöser Programmfehler schauen Sie bitte in die Log-Datei!

6.5.5 Projekt speichern

Das Projekt wird bei jeder Änderung automatisch gespeichert, so dass es dafür keinen Befehl gibt.

6.6 Die Programmiersprache C

Der SDCC-Compiler enthält eine 80-seitige Dokumentation auf Englisch. Diese liegt als PDF-Datei im Ordner „sdcc\doc“. Die wichtigsten Aspekte werden in diesem Handbuch auf Deutsch besprochen. Fortgeschrittene Anwender sollten sich die Originaldokumentation aber zu Gemüte führen. Wichtige Auszüge darauf finden Sie ab Seite 96.

Der Compiler versteht ANSI-C 95 mit einigen Ergänzungen für den Betrieb auf Mikrocontrollern. Eine kostengünstige Dokumentation des ANSI-C Sprachumfanges und der Standardbibliotheken ist im Verlag O’Reilly erschienen:

Peter Prinz und Ulla Kirch-Prinz, C kurz & gut, O’Reillys Taschenbibliothek, 120 Seiten, 8 Euro, in jeder Buchhandlung.

6.7 Spracherweiterungen und Besonderheiten

An dieser Stelle sollen die Erweiterungen und Besonderheiten des SDCC-Compilers beschrieben werden, die über den Sprachumfang von ANSI-C hinausgehen, bzw. die dazu im Widerspruch stehen.

6.7.1 Speicherklassen

`data / near`

Dies ist die Vorgabe für die Benutzung des „small“-Speichermodells. Daten, die als *data* (oder, synonym, *near*) deklariert wurden, finden im internen Speicher Platz.

Beispiel:

```
data unsigned char test_data;
```

`xdata / far`

Umgekehrt ist dies die Vorgabe für das „large”-Speichermodell. Daten, die als *xdata* oder *far* deklariert wurden, werden im XRAM abgelegt.

Beispiel:

```
xdata unsigned char test_Xdata;
```

`idata`

Variablen, die als idata deklariert wurden, platziert der Linker im indirekt adressierbaren internen Datenspeicher.

code

Diese Speicherklasse ist praktisch, wenn Konstanten im Code-Speicher abgelegt werden sollen.

Beispiel:

```
code char test_vektor[] = {'T', 'e', 's', 't');
```

bit

Dies ist zugleich eine Deklaration von Datentyp und Speicherklasse. Eine als bit deklarierte Variable wird im bit-adressierbaren Speicher des Controllers abgelegt.

Beispiel:

```
bit test_bit;
test_bit = 1;
if (test_bit == 1) then ...
```

sfr / sbit

Zugriff auf Special Function Registers und zugehörige Bit-Variablen.

Beispiel:

```
sfr at 0x80 P0;
sbit at 0xd7 CY;
```

6.7.2 Zeiger

Werden Zeiger benutzt, so muss sichergestellt werden, dass der Zeiger auf das richtige Speichersegment zeigt. Dies geschieht dadurch, dass ein normaler („generischer“) Zeiger aus drei Bytes besteht, von denen ein Byte spezifiziert, auf welches Segment die Adresse zeigt. Um einen effizienteren Code zu erzeugen, kann dem Zeiger aber auch explizit sein Speichersegment zugewiesen werden.

Ein generischer Zeiger wird wie folgt deklariert:

```
char * xdata p;
```

Der Zeiger p liegt physikalisch im XRAM-Segment. Er kann, je nach seiner späteren Zuweisung, auf beliebige Datentypen in beliebigen Speichersegmenten zeigen.

Wird der Deklaration eine weitere Speicherklassen-Deklaration vorangestellt, so hat man keinen generischen Zeiger mehr, sondern der Zeiger weist explizit in das angegebene Segment:

```
data char * xdata p;
```

deklariert einen Zeiger, der auf eine 8-bit Datenstruktur im Data-Segment zeigt. Der Zeiger selbst ist im XRAM gespeichert.

6.7.3 Absolute Adressierung

Neben der Adressierung von SFR interessiert den Mikrocontroller-Programmierer natürlich besonders, wie Speicherstellen im XRAM angesprochen werden können, um z.B. Memory-Mapped Peripherie zu steuern. Zu diesem Zweck dient das reservierte Wort „*at*“.

Beispiel:

```
xdata at 0x7ffe unsigned int chksum;
```

Die 16-bit Variable “*chksum*” wird ab Adresse 7FFEh im XRAM abgelegt. Der Compiler reserviert freilich keinen Platz für diese Adresse; sie ist wie ein EQU-Kommando im Assembler zu sehen.

Für den Fall, dass sich der Inhalt der Variable durch externe Ereignisse ändern kann, sollte eine Deklaration als „*volatile*“ erfolgen:

```
volatile xdata at 0xf401 unsigned int lcd_status;
```

Damit optimiert der Compiler wiederholte Zugriffe auf eine Speicherstelle nicht weg. *Volatile*-Deklarationen sind natürlich auch nützlich, wenn es um die Programmierung von Interrupts geht, und eine Interrupt-Service-Routine eine globale Variable verändert.

Natürlich gilt die absolute Adressierung auch für Bit-Variablen.

6.7.4 Parameter, lokale Variablen und reentrante Funktionen

Im Unterschied zu üblichen C-Compilern für PC-Programmierung o.ä. werden automatische (lokale) Variablen und Parameter vorzugsweise im Datensegment abgelegt, und zwar je nach Speichermodell im internen oder externen Datenspeicher. Dies entspricht einer Deklaration als „static“, d.h. eine Funktion kann sich zwischen zwei Aufrufen eine Variable merken, da diese statisch an die Funktion gebunden ist und nie freigegeben wird.

Soll jedoch eine Funktion reentrant sein, d.h. sie ruft sich selbst auf oder wird möglicherweise durch eine Interrupt-Service-Routine aufgerufen (d.h. der Prozessor befindet sich quasi simultan mehrere Male in dieser Routine), so muss dafür gesorgt werden, dass lokale Variablen und Parameter im Stack gespeichert werden. Dies geschieht über das reservierte Wort reentrant:

```
unsigned char foo(char i) reentrant
{
  int j = 0;
  static data at 0x31 unsigned char k;
  ...
}
```

Hier wird für jeden Aufruf der Funktion neuer Platz für die übergebene Variable i und die lokale Variable j geschaffen. Die Variable k liegt an einer bestimmten Adresse und muss statisch gebunden werden.

Es ist unnötig zu erwähnen, dass der Platz auf dem Stack knapp ist und reentrante Funktionen nur sparsam verwendet werden sollten. An die Programmierung rekursiver Funktionen zu denken verbietet sich von selbst.

Parameter dürfen niemals als bestimmte Speicherklasse übergeben werden. Ihr Datentyp ergibt sich aus dem verwendeten Speichermodell und aus den Reentranz-Optionen.

6.7.5 Overlaying

Um Speicherplatz im internen RAM zu sparen, versucht der Compiler, lokale Variablen, die keiner Speicherklasse zugewiesen wurden, und Parameter wiederzuverwenden. Dazu werden diese Daten in einem Speicherbereich abgelegt, der überlagert werden kann („*Overlaying*“). Ein Aufruf einer anderen Routine überlagert dann diese Daten. Dies gilt nur für das „small“-Speichermodell! Es schränkt das oben gesagte über statisch gebundene Variablen ein. Eine echte statische Bindung ohne *Overlaying* erfolgt daher nur mit dem reservierten Wort „*static*“.

Über die Direktive „*#pragma nooverlay*“ kann das *Overlaying* abgeschaltet werden. Beispiel:

```
#pragma save
#pragma nooverlay
void set_error(unsigned char errcd)
{
  P3 = errcd;
}
#pragma restore

void some_isr () interrupt 2
{
  ...
set_error(10);
  ...
}
```

Ohne die *#pragma*-Direktive könnte es sein, dass der Parameter *errcd* überschrieben würde, und der Aufruf durch die Interrupt-Service-Routine würde zu einem erratischen Laufzeitverhalten des Programms führen.

6.7.6 Interrupt-Service-Routinen

Kommen wir jetzt zu den schon angesprochenen Interrupt-Service-Routinen (ISR). Diese sind in C besonders einfach zu programmieren, da sich der Programmierer weder um das Stacken von Registern noch um die Interrupt-Vektoren kümmern muss.

Beispiel:

```
void timer_isr(void) interrupt 1 using 2
{
  ...
}
```

Die Zahl hinter dem Wort „*interrupt*“ bezeichnet die Interrupt-Nummer, die die ISR bedient. Das Wort „*using*“ bezeichnet die verwendete Registerbank. Um die Freigabe der Interrupts und deren Prioritätsschema muss sich der Programmierer aber noch selbst kümmern. Die Registerbanken sollten sorgfältig vergeben werden.

Interrupt-Service-Routinen öffnen Tür und Tor für einige hochinteressante Bugs:

- Wenn die ISR Variablen ändert, auf die andere Routinen zugreifen, dann sollten diese Variablen als *volatile* deklariert werden.
- Wenn der Zugriff auf solche Variablen nicht atomar ist, d.h. er besteht aus mehr als einem einzigen Maschinenzyklus, dann sollte der betreffende Interrupt maskiert (*disabled*) werden, während auf die Variable zugegriffen wird. Dies ist schwer einzuschätzen, da nicht klar ist, wie der Compiler einen solchen Zugriff übersetzt.
- Fließkomma-Arithmetik und *int* (16-bit) und *long* (32-bit)-Multiplikation/Division/Modulo-Operatinen sind als externe ANSI-C Programme implementiert. Solche Aufrufe sollten wenn möglich nicht von ISR getätigt werden. Mehr hierzu in der Original SDCC Dokumentation.
- Aufrufe von anderen Funktionen aus einer ISR sollten vermieden werden. Ist das nicht möglich, so sollte die Funktion *reentrant* dekariert oder mit *#pragma nooverlay* versehen werden.

Wird die ISR ohne das „*using*“-Argument aufgerufen, so speichert sie den Registersatz auf dem Stack. Verwendet man das „*using*“-Argument, so wird die angegebene Registerbank zerstört.

6.7.7 Kritische Routinen

Um bequem Routinen vor dem Zerschießen durch eine ISR zu schützen, gibt es die „*critical*“-Umgebung. Sie blockiert alle Interrupts vor dem Aufruf und stellt den alten Zustand nach Verlassen der Routine wieder her.

Beispiel:

```
int foo() critical
{
  ...
  ...
}
```

Das Attribut „*critical*” kann mit anderen Attributen wie *reentrant* kombiniert werden. Es kann ferner auch lokal angewandt werden:

```
critical{ i++; }
```

6.7.8 Startup Code

Der Compiler fügt einen Aufruf der Routine „*_sdcc_external_startup()*“ am Beginn des Codes ein. Diese Routine enthält normalerweise keinen Code sondern gibt nur eine 0 zurück. Für die Verwendung mit CESY wurde die Routine angepasst; sie setzt das Prozessor-Status-Word PSW auf 0 um zur Registerbank 0 zu schalten.

Die Routine „*_sdcc_external_startup()*“ kann vom Programm überschrieben werden, um weitere Initialisierungen vorzunehmen, bevor die Standard-Initialisierung der Datenbereiche erfolgt. Denken Sie aber daran, immer als erstes einen psw=0 Befehl einzufügen!

Wenn die Routine etwas anderes als Null zurückgibt, so wird die standardmäßige Variablen-Initialisierung übersprungen.

6.7.9 Integer-Arithmetik

Der Compiler verwendet drei Integer-Typen:

- char: 8-bit. Nicht nötigerweise ein ASCII-Zeichen.
- int: 16-bit
- long: 32-bit

Alle Typen gibt es mit oder ohne Vorzeichen (unsigned). Multiplikation und Division bei den 16- und 32-bit Typen erfolgen über externe C-Funktionen, die, wie bereits beschrieben, als nicht reentrant übersetzt sind. Näheres hierzu siehe die SDCC-Originaldokumentation.

6.7.10 Float-Arithmetik

SDCC kann mit Fließkommazahlen arbeiten; diese Eigenschaft ist aber noch experimentell. Die nötige Standard-Bibliothek „float“ ist in C programmiert und ist daher nicht so schnell wie sie sein könnte, wäre sie in Assembler programmiert. Außerdem ist zu beachten, dass die Bibliothek für das richtige Speichermodell übersetzt wird.

Wir haben die Fließkomma-Arithmetik im „small“-Speichermodell getestet, ohne dass irgendetwas neu übersetzt werden musste. Leider funktioniert die Ausgabe per printf/sprintf-Befehl noch nicht.

6.8 Weitere Dokumentation

Weitere Dokumentation, z.B. zur Inline-Assembler-Programmierung und zu den Standardbibliotheken, gibt es in der englischsprachigen Original-Anleitung zum SDCC, in Auszügen ab Seite 96 in diesem Handbuch.

7 Diverses

7.1 Eigenschaften

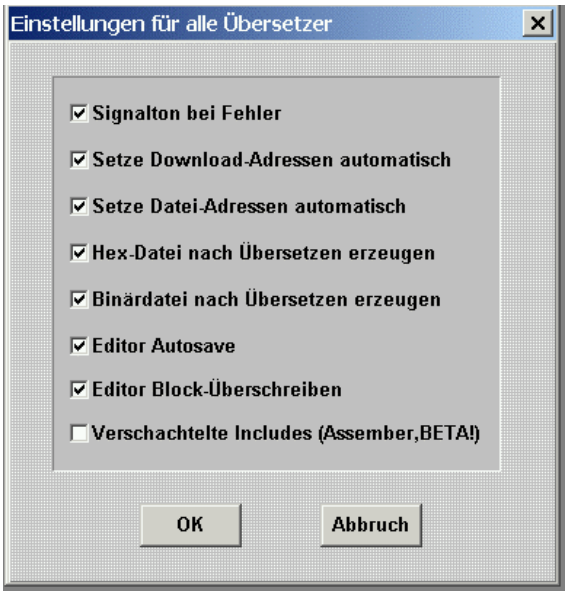


Abbildung 23: Einstellungen für alle Übersetzer

Im Menü „Optionen—Eigenschaften“ können Sie einige automatische Funktionen konfigurieren, die Ihnen das Leben etwas erleichtern sollen.

Die Adressbereiche für den Download oder für das Speichern des erzeugten Codes können automatisch auf Start- und Endadresse des Assemblier-, bzw. Compilierablaufes gesetzt werden.

Nach dem Assemblieren/Compilieren kann wahlweise direkt eine Hex-/Binärdatei erzeugt werden. Diese finden Sie in dem Ordner, in dem auch der Quelltext steht.

Editor-Autosave sorgt dafür, dass Ihre Texte vor dem Assemblieren auf Platte gespeichert werden. Im Falle eines CESY-Absturzes verlieren Sie keine Daten.

Wenn Sie im Editor einen Block markiert haben, wird der ganze Block durch einen Tastenanschlag gelöscht. Dies können Sie in den Optionen abschalten, um Sicherheit vor Datenverlust zu haben.

Wieviel Sinn Include-Dateien machen, die weitere Include-Dateien aufrufen, bleibt Ihnen überlassen. Wenn Sie das rekursive Einbinden erlauben wollen, müssen Sie hier das entsprechende Häkchen setzen. Diese Funktion befindet sich aber Beta-Stadium. Dies wird sich auch erst mal nicht ändern.

7.2 Blitzstart

Abbildung 24: Blitzstart-Knöpfe

Über die Blitzstart-Knöpfe (siehe Abbildung) können Sie den Arbeitsgang von Assemblieren – Download – Programmstart automatisieren. Sie müssen dazu nur auf den entsprechenden Blitzstart-Knopf klicken. Vorher muss aber in den Optionen—Eigenschaften das Kästchen „Download-Adressen setzen“ aktiviert werden.

Entsprechendes gilt natürlich auch für die Compiler.

Wenn Sie in den Terminal-Optionen „Automatisch Starten“ aktiviert haben, so geht sofort nach dem Starten ein Terminalfenster auf, bzw. wenn schon eins offen ist,

wird es aktiviert. Beachten Sie aber, dass das Öffnen eines Terminalfensters mit dem Initialisieren einer Schnittstelle verbunden ist, was ein paar hundert Millisekunden dauert.

7.3 Neue SDCC-Version

Auf der Website http://sdcc.sourceforge.net können Sie laufend aktualisierte SDCC-Versionen herunterladen. Installieren Sie diese, wie oben beschrieben. Vergessen Sie nicht, die CESY-Startup-Dateien neu zu installieren!

7.4 Bug gefunden?

Leider kann es nicht ausgeschlossen werden, dass ein ausgeliefertes Programm noch Bugs enthält. Falls Sie einen solchen finden, bitten wir Sie, uns diesen *schriftlich* (per Post oder e-mail) zu melden. Wir werden den Fehler schnellstmöglich beheben und Ihnen eine korrigierte Version kostenlos zusenden. Bitte geben Sie folgende Informationen an, damit wir den Fehler reproduzieren können:

- Name, Adresse, Kunden-Nummer, Kaufdatum
- Seriennummer
- Betriebssystem
- Controllersystem, evtl. Schaltplan
- CESY-Händler
- Controllersystem-Händler
- evtl. Schaltplan des seriellen Kabels
- detaillierte Beschreibung des Fehlers.

8 Der Befehlssatz des 80x51

Die Assembler-Befehle sind für alle Prozessoren der 80x51er Familie identisch.

8.1.1 Transferbefehle für Register und internes RAM

```
MOV  A,Rr                    MOV   Rr,A
MOV  A,@Ri                   MOV   @Ri,A
MOV  A,dadr                  MOV   dadr,A
MOV  A,#konst8               MOV   dadr,#konst8
MOV  Rr,#konst8              MOV   @Ri,#konst8
MOV  Rr,dadr                 MOV   dadr,Rr
MOV  @Ri,dadr                MOV   dadr,@Ri
MOV  dadr1,dadr2
MOV  DPTR,#konst16
XCH A,Rr      XCH  A,@Ri
XCH A,dadr
XCHD     A,@Ri

PUSH     dadr POP  dadr
```

8.1.2 Transferbefehle für externes RAM und ROM

```
MOVX   A,@Ri                 MOVX    @Ri,A
MOVX   A,@DPTR               MOVX    @DPTR,A
                             MOVC A,@A+PC
MOVC     A,@A+DPTR
```

8.1.3 Logik-Befehle

```
ANL A,Rr                     ANL  A,@Ri
ANL A,dadr                   ANL  dadr,A
ANL A,#konst8                ANL  dadr,#konst8

ORL A,Rr                     ORL  A,@Ri
ORL A,dadr                   ORL  dadr,A
ORL A,#konst8                ORL  dadr,#konst8

XRL A,Rr                     XRL  A,@Ri
XRL A,dadr                   XRL  dadr,A
XRL A,#konst8                XRL  dadr,#konst8
                             CLR  A
CPL A
```

8.1.4 Bit-Verarbeitung

```
MOV C,badr                   MOV  badr,C
ANL C,badr                   ANL  C,/badr
ORL C,badr                   ORL  C,/badr

CLR   C                      CLR  badr
SETB  C                      SETB badr
CPL   C                      CPL  badr
```

8.1.5 Arithmetik-Befehle

```
ADD A,Rr                    ADD  A,@Ri
ADD A,dadr                  ADD  A,#konst8

ADDC     A,Rr               ADDC A,@Ri
ADDC     A,dadr             ADDC A,#konst8

INC A                       INC  dadr
INC Rr                      INC  @Rr
INC DPTR

SUBB     A,Rr               SUBB A,@Ri
SUBB     A,dadr             SUBB A,#konst8

DEC A                       DEC  dadr
DEC Rr                      DEC  @Rr

MUL AB
DIV AB

DA  A
```

8.1.6 Rotations-Befehle

```
RL  A                       RR   A
RLC A                       RRC  A
```

8.1.7 Unbedingte Sprungbefehle

```
LJMP     adr16              AJMP adr11
SJMP     rel                JMP  @A+DPTR
```

8.1.8 Bedingte Sprungbefehle

```
JZ  rel                     JNZ  rel
JC  rel                     JNC  rel
JB  badr,rel                JNB  badr,rel
JBC badr,rel
```

8.1.9 Kombinierte Vergleichs- und Sprungbefehle

```
CJNE     A,dadr,rel         CJNE A,#konst8,rel
CJNE     Rr,#konst8,rel     CJNE @Ri,#konst8,rel
```

8.1.10 Kombinierte Dekrementier- und Sprungbefehle

```
DJNZ     Rr,rel             DJNZ dadr,rel
```

8.1.11 Unterprogramm-Befehle

```
LCALL adr16                 ACALL     adr11
RET                         RETI
```

8.1.12 Diverse Befehle

```
NOP                         SWAP A
```

Auszüge der
Wichtigsten Kapitel
aus dem
Original SDCC-Handbuch

Chapter 3

Using SDCC

3.1 Compiling

3.1.1 Single Source File Projects

For single source file 8051 projects the process is very simple. Compile your programs with the following command **"sdcc sourcefile.c".** This will compile, assemble and link your source file. Output files are as follows:

- sourcefile.asm - Assembler source file created by the compiler
- sourcefile.lst - Assembler listing file created by the Assembler
- sourcefile.rst - Assembler listing file updated with linkedit information, created by linkage editor
- sourcefile.sym - symbol listing for the sourcefile, created by the assembler
- sourcefile.rel or sourcefile.o - Object file created by the assembler, input to Linkage editor
- sourcefile.map - The memory map for the load module, created by the Linker
- sourcefile.mem - A file with a summary of the memory usage
- sourcefile.ihx - The load module in Intel hex format (you can select the Motorola S19 format with --out-fmt-s19. If you need another format you might want to use *objdump* or *srecord*). Both formats are documented in the documentation of srecord
- sourcefile.adb - An intermediate file containing debug information needed to create the .cdb file (with --debug)
- sourcefile.cdb - An optional file (with --debug) containing debug information. The format is documented in cdbfileformat.pdf.
- sourcefile. - (no extension) An optional AOMF or AOMF51 file containing debug information (generated with option --debug). The (Intel) *a*bsolute *o*bject *m*odule *f*ormat is commonly used by third party tools (debuggers, simulators, emulators)
- sourcefile.dump* - Dump file to debug the compiler it self (generated with option --dumpall) (see section 3.2.9 and section 9.1 "Anatomy of the compiler").

3.1.2 Projects with Multiple Source Files

SDCC can compile only ONE file at a time. Let us for example assume that you have a project containing the following files:

foo1.c (contains some functions)
foo2.c (contains some more functions)
foomain.c (contains more functions and the function main)

The first two files will need to be compiled separately with the commands:

sdcc -c foo1.c
sdcc -c foo2.c

Then compile the source file containing the *main()* function and link the files together with the following command:

sdcc foomain.c foo1.rel foo2.rel

Alternatively, *foomain.c* can be separately compiled as well:

sdcc -c foomain.c
sdcc foomain.rel foo1.rel foo2.rel

The file containing the *main()* function MUST be the FIRST file specified in the command line, since the linkage editor processes file in the order they are presented to it. The linker is invoked from SDCC using a script file with extension .lnk. You can view this file to troubleshoot linking problems such as those arising from missing libraries.

3.1.3 Projects with Additional Libraries

Some reusable routines may be compiled into a library, see the documentation for the assembler and linkage editor (which are in <installdir>/share/sdcc/doc) for how to create a *.lib* library file. Libraries created in this manner can be included in the command line. Make sure you include the -L <library-path> option to tell the linker where to look for these files if they are not in the current directory. Here is an example, assuming you have the source file *foomain.c* and a library *foolib.lib* in the directory *mylib* (if that is not the same as your current project):

sdcc foomain.c foolib.lib -L mylib

Note here that *mylib* must be an absolute path name.

The most efficient way to use libraries is to keep separate modules in separate source files. The lib file now should name all the modules.rel files. For an example see the standard library file *libsdcc.lib* in the directory <installdir>/share/lib/small.

3.1.4 Using sdcclib to Create and Manage Libraries

Alternatively, instead of having a .rel file for each entry on the library file as described in the preceding section, sdcclib can be used to embed all the modules belonging to such library in the library file itself. This results in a larger library file, but it greatly reduces the number of disk files accessed by the linker. Additionally, the packed library file contains an index of all include modules and symbols that significantly speeds up the linking process. To display a list of options supported by sdcclib type:

sdcclib -?

To create a new library file, start by compiling all the required modules. For example:

sdcc -c _divsint.c
sdcc -c _divuint.c
sdcc -c _modsint.c
sdcc -c _moduint.c
sdcc -c _mulint.c

This will create files _divsint.rel, _divuint.rel, _modsint.rel, _moduint.rel, and _mulint.rel. The next step is to add the .rel files to the library file:

sdcclib libint.lib _divsint.rel
sdcclib libint.lib _divuint.rel
sdcclib libint.lib _modsint.rel
sdcclib libint.lib _moduint.rel
sdcclib libint.lib _mulint.rel

If the file already exists in the library, it will be replaced. To see what modules and symbols are included in the library, options -s and -m are available. For example:

sdcclib -s libint.lib
_divsint.rel:
__divsint_a_1_1
__divsint_PARM_2
__divsint
_divuint.rel:
__divuint_a_1_1
__divuint_PARM_2
__divuint_reste_1_1
__divuint_count_1_1
__divuint
_modsint.rel:
__modsint_a_1_1
__modsint_PARM_2
__modsint
_moduint.rel:
__moduint_a_1_1
__moduint_PARM_2
__moduint_count_1_1
__moduint
_mulint.rel:
__mulint_PARM_2
__mulint

If the source files are compiled using –debug, the corresponding debug information file .adb will be include in the library file as well. The library files created with sdcclib are plain text files, so they can be viewed with a text editor. It is not recomended to modify a library file created with sdcclib using a text editor, as there are file indexes numbers located accross the file used by the linker to quickly locate the required module to link. Once a .rel file (as well as a .adb file) is added to a library using sdcclib, it can be safely deleted, since all the information required for linking is embedded in the library file itself. Library files created using sdcclib are used as described in the preceding sections.

3.2 Command Line Options

3.2.1 Processor Selection Options

-mmcs51 Generate code for the Intel MCS51 family of processors. This is the default processor target.

-mds390 Generate code for the Dallas DS80C390 processor.

-mds400 Generate code for the Dallas DS80C400 processor.

-mhc08 Generate code for the Motorola HC08 family of processors (added Oct 2003).

-mz80 Generate code for the Zilog Z80 family of processors.

-mgbz80 Generate code for the GameBoy Z80 processor (Not actively maintained).

-mavr Generate code for the Atmel AVR processor (In development, not complete). AVR users should probably have a look at avr-gcc http://savannah.nongnu.org/download/avr-libc/snapshots/.

-mpic14 Generate code for the Microchip PIC 14-bit processors (p16f84 and variants. In development, not complete).

-mpic16 Generate code for the Microchip PIC 16-bit processors (p18f452 and variants. In development, not complete).

-mtlcs900h Generate code for the Toshiba TLCS-900H processor (Not maintained, not complete).

-mxa51 Generate code for the Phillips XA51 processor (Not maintained, not complete).

3.2.2 Preprocessor Options

-I<path> The additional location where the pre processor will look for <..h> or "..h" files.

-D<macro[=value]> Command line definition of macros. Passed to the preprocessor.

-M Tell the preprocessor to output a rule suitable for make describing the dependencies of each object file. For each source file, the preprocessor outputs one make-rule whose target is the object file name for that source file and whose dependencies are all the files '#include'd in it. This rule may be a single line or may be continued with '\'-newline if it is long. The list of rules is printed on standard output instead of the preprocessed C program. '-M' implies '-E'.

-C Tell the preprocessor not to discard comments. Used with the '-E' option.

-MM Like '-M' but the output mentions only the user header files included with '#include "file"'. System header files included with '#include <file>' are omitted.

-Aquestion(answer) Assert the answer answer for question, in case it is tested with a preprocessor conditional such as '#if #question(answer)'. '-A-' disables the standard assertions that normally describe the target machine.

-Umacro Undefine macro macro. '-U' options are evaluated after all '-D' options, but before any '-include' and '-imacros' options.

-dM Tell the preprocessor to output only a list of the macro definitions that are in effect at the end of preprocessing. Used with the '-E' option.

-dD Tell the preprocessor to pass all macro definitions into the output, in their proper sequence in the rest of the output.

-dN Like '-dD' except that the macro arguments and contents are omitted. Only '#define name' is included in the output.

-Wp preprocessorOption[,preprocessorOption]... Pass the preprocessorOption to the preprocessor `sdcpp`. SDCC uses an adapted version of the preprocessor cpp of the GNU Compiler Collection (gcc), if you need more dedicated options please refer to the documentation at http://www.gnu.org/software/gcc/onlinedocs/.

3.2.3 Linker Options

-L --lib-path <absolute path to additional libraries> This option is passed to the linkage editor's additional libraries search path. The path name must be absolute. Additional library files may be specified in the command line. See section Compiling programs for more details.

--xram-loc <Value> The start location of the external ram, default value is 0. The value entered can be in Hexadecimal or Decimal format, e.g.: --xram-loc 0x8000 or --xram-loc 32768.

--code-loc <Value> The start location of the code segment, default value 0. Note when this option is used the interrupt vector table is also relocated to the given address. The value entered can be in Hexadecimal or Decimal format, e.g.: --code-loc 0x8000 or --code-loc 32768.

--stack-loc <Value> By default the stack is placed after the data segment. Using this option the stack can be placed anywhere in the internal memory space of the 8051. The value entered can be in Hexadecimal or Decimal format, e.g. --stack-loc 0x20 or --stack-loc 32. Since the sp register is incremented before a push or call, the initial sp will be set to one byte prior the provided value. The provided value should not overlap any other memory areas such as used register banks or the data segment and with enough space for the current application.

--data-loc <Value> The start location of the internal ram data segment. The value entered can be in Hexadecimal or Decimal format, eg. --data-loc 0x20 or --data-loc 32. (By default, the start location of the internal ram data segment is set as low as possible in memory, taking into account the used register banks and the bit segment at address 0x20. For example if register banks 0 and 1 are used without bit variables, the data segment will be set, if --data-loc is not used, to location 0x10.)

--idata-loc <Value> The start location of the indirectly addressable internal ram of the 8051, default value is 0x80. The value entered can be in Hexadecimal or Decimal format, eg. --idata-loc 0x88 or --idata-loc 136.

--bit-loc <Value> The start location of the bit addressable internal ram of the 8051. This is *not* implemented yet. Instead an option can be passed directly to the linker: -Wl -bBSEG=<Value>.

--out-fmt-ihx The linker output (final object code) is in Intel Hex format. This is the default option. The format itself is documented in the documentation of srecord.

--out-fmt-s19 The linker output (final object code) is in Motorola S19 format. The format itself is documented in the documentation of srecord.

-Wl linkOption[,linkOption]... Pass the linkOption to the linker. See file sdcc/as/doc/asxhtm.html for more on linker options.

3.2.4 MCS51 Options

--model-small Generate code for Small Model programs, see section Memory Models for more details. This is the default model.

--model-large Generate code for Large model programs, see section Memory Models for more details. If this option is used all source files in the project have to be compiled with this option.

--xstack Uses a pseudo stack in the first 256 bytes in the external ram for allocating variables and passing parameters. See section 3.17.1.2 External Stack for more details.

--iram-size <Value> Causes the linker to check if the internal ram usage is within limits of the given value.

--xram-size <Value> Causes the linker to check if the external ram usage is within limits of the given value.

--code-size <Value> Causes the linker to check if the code memory usage is within limits of the given value.

--stack-size <Value> Causes the linker to check if there is at minimum <Value> bytes for stack.

--pack-iram Causes the linker use unused register banks for data variables or stack.

3.2.5 DS390 / DS400 Options

--model-flat24 Generate 24-bit flat mode code. This is the one and only that the ds390 code generator supports right now and is default when using *-mds390*. See section Memory Models for more details.

--protect-sp-update disable interrupts during ESP:SP updates.

--stack-10bit Generate code for the 10 bit stack mode of the Dallas DS80C390 part. This is the one and only that the ds390 code generator supports right now and is default when using *-mds390*. In this mode, the stack is located in the lower 1K of the internal RAM, which is mapped to 0x400000. Note that the support is incomplete, since it still uses a single byte as the stack pointer. This means that only the lower 256 bytes of the potential 1K stack space will actually be used. However, this does allow you to reclaim the precious 256 bytes of low RAM for use for the DATA and IDATA segments. The compiler

will not generate any code to put the processor into 10 bit stack mode. It is important to ensure that the processor is in this mode before calling any re-entrant functions compiled with this option. In principle, this should work with the *--stack-auto* option, but that has not been tested. It is incompatible with the *--xstack* option. It also only makes sense if the processor is in 24 bit contiguous addressing mode (see the *--model-flat24 option*).

--stack-probe insert call to function __stack_probe at each function prologue.

--tini-libid <nnnn> LibraryID used in -mTININative.

--use-accelerator generate code for DS390 Arithmetic Accelerator.

3.2.6 Z80 Options

--callee-saves-bc Force a called function to always save BC.

--no-std-crt0 When linking, skip the standard crt0.o object file. You must provide your own crt0.o for your system when linking.

3.2.7 Optimization Options

--nogcse Will not do global subexpression elimination, this option may be used when the compiler creates undesirably large stack/data spaces to store compiler temporaries. A warning message will be generated when this happens and the compiler will indicate the number of extra bytes it allocated. It is recommended that this option NOT be used, #pragma nogcse can be used to turn off global subexpression elimination for a given function only.

--noinvariant Will not do loop invariant optimizations, this may be turned off for reasons explained for the previous option. For more details of loop optimizations performed see section Loop Invariants. It is recommended that this option NOT be used, #pragma noinvariant can be used to turn off invariant optimizations for a given function only.

--noinduction Will not do loop induction optimizations, see section strength reduction for more details. It is recommended that this option is NOT used, #pragma noinduction can be used to turn off induction optimizations for a given function only.

--nojtbound Will not generate boundary condition check when switch statements are implemented using jumptables. See section 8.1.7 Switch Statements for more details. It is recommended that this option is NOT used, #pragma nojtbound can be used to turn off boundary checking for jump tables for a given function only.

--nooopreverse Will not do loop reversal optimization.

--**nolabelopt** Will not optimize labels (makes the dumpfiles more readable).

--no-xinit-opt Will not memcpy initialized data from code space into xdata space. This saves a few bytes in code space if you don't have initialized data.

--nooverlay The compiler will not overlay parameters and local variables of any function, see section Parameters and local variables for more details.

--no-peep Disable peep-hole optimization.

--peep-file <filename> This option can be used to use additional rules to be used by the peep hole optimizer. See section 8.1.12 Peep Hole optimizations for details on how to write these rules.

--peep-asm Pass the inline assembler code through the peep hole optimizer. This can cause unexpected changes to inline assembler code, please go through the peephole optimizer rules defined in the source file tree '<target>/peeph.def' before using this option.

3.2.8 Other Options

-c --compile-only will compile and assemble the source, but will not call the linkage editor.

--c1mode reads the preprocessed source from standard input and compiles it. The file name for the assembler output must be specified using the -o option.

-E Run only the C preprocessor. Preprocess all the C source files specified and output the results to standard output.

-o <path/file> The output path resp. file where everything will be placed. If the parameter is a path, it must have a trailing slash (or backslash for the Windows binaries) to be recognized as a path.

--stack-auto All functions in the source file will be compiled as *reentrant*, i.e. the parameters and local variables will be allocated on the stack. see section Parameters and Local Variables for more details. If this option is used all source files in the project should be compiled with this option.

--callee-saves function1[,function2][,function3].... The compiler by default uses a caller saves convention for register saving across function calls, however this can cause unnecessary register pushing & popping when calling small functions from larger functions. This option can be used to switch the register saving convention for the function names specified. The compiler will not save registers when calling these functions, no extra code will be generated at the entry & exit (function prologue & epilogue) for these functions to save & restore the registers used by these functions, this can SUBSTANTIALLY reduce code & improve run time performance of the generated code. In the future the compiler (with inter procedural analysis) will be able to determine the appropriate scheme to use for each function call. DO NOT use this option for built-in functions such as _mulint..., if this option is used for a library function the appropriate library function needs to be recompiled with the same option. If the project consists of multiple source files then all the source file should be compiled with the same --callee-saves option string. Also see #pragma callee_saves.

--debug When this option is used the compiler will generate debug information. The debug information collected in a file with .cdb extension can be used with the SDCDB. For more information see documentation for SDCDB. Another file with no extension contains debug information in AOMF or AOMF51 format which is commonly used by third party tools.

-S Stop after the stage of compilation proper; do not assemble. The output is an assembler code file for the input file specified.

--int-long-reent Integer (16 bit) and long (32 bit) libraries have been compiled as reentrant. Note by default these libraries are compiled as non-reentrant. See section Installation for more details.

--cyclomatic This option will cause the compiler to generate an information message for each function in the source file. The message contains some *important* information about the function. The number of edges and nodes the compiler detected in the control flow graph of the function, and most importantly the *cyclomatic complexity* see section on Cyclomatic Complexity for more details.

--float-reent Floating point library is compiled as reentrant. See section Installation for more details.

--main-return This option can be used when the code generated is called by a monitor program. The compiler will generate a 'ret' upon return from the 'main' function. The default setting is to lock up i.e. generate a '`sjmp .`'.

--nostdincl This will prevent the compiler from passing on the default include path to the preprocessor.

--nostdlib This will prevent the compiler from passing on the default library path to the linker.

--verbose Shows the various actions the compiler is performing.

-V Shows the actual commands the compiler is executing.

--no-c-code-in-asm Hides your ugly and inefficient c-code from the asm file, so you can always blame the compiler :).

--i-code-in-asm Include i-codes in the asm file. Sounds like noise but is most helpful for debugging the compiler itself.

--less-pedantic Disable some of the more pedantic warnings (jwk burps: please be more specific here, please!). If you want rather more than less warnings you should consider using a separate tool dedicated to syntax checking like splint www.splint.org.

--print-search-dirs Display the directories in the compiler's search path

--vc Display errors and warnings using MSVC style, so you can use SDCC with visual studio.

--use-stdout Send errors and warnings to stdout instead of stderr.

-Wa asmOption[,asmOption]... Pass the asmOption to the assembler. See file sdcc/as/doc/asxhtm.html for assembler options.

3.2.9 Intermediate Dump Options

The following options are provided for the purpose of retargetting and debugging the compiler. These provided a means to dump the intermediate code (iCode) generated by the compiler in human readable form at various stages of the compilation process. More on iCodes see chapter 9.1 "The anatomy of the compiler".

--dumpraw This option will cause the compiler to dump the intermediate code into a file of named <*source filename*>.*dumpraw* just after the intermediate code has been generated for a function, i.e. before any optimizations are done. The basic blocks at this stage ordered in the depth first number, so they may not be in sequence of execution.

--dumpgcse Will create a dump of iCode's, after global subexpression elimination, into a file named <*source filename*>.*dumpgcse.*

--dumpdeadcode Will create a dump of iCode's, after deadcode elimination, into a file named <*source filename*>.*dumpdeadcode.*

--dumploop Will create a dump of iCode's, after loop optimizations, into a file named <*source filename*>.*dumploop.*

--dumprange Will create a dump of iCode's, after live range analysis, into a file named <*source filename*>.*dumprange.*

--dumlrange Will dump the life ranges for all symbols.

--dumpregassign Will create a dump of iCode's, after register assignment, into a file named <*source filename*>.*dumprassgn.*

--dumplrange Will create a dump of the live ranges of iTemp's

--dumpall Will cause all the above mentioned dumps to be created.

3.2.10 Redirecting output on Windows Shells

By default SDCC writes it's error messages to "standard error". To force all messages to "standard output" use **--**use-stdout. Additionally, if you happen to have visual studio installed in your windows machine, you can use it to compile your sources using a custom build and the SDCC --vc option. Something like this should work:

c:\sdcc\bin\sdcc.exe --vc --model-large -c $(InputPath)

3.3 Environment variables

SDCC recognizes the following environment variables:

SDCC_LEAVE_SIGNALS SDCC installs a signal handler to be able to delete temporary files after an user break (^C) or an exception. If this environment variable is set, SDCC won't install the signal handler in order to be able to debug SDCC.

TMP, TEMP, TMPDIR Path, where temporary files will be created. The order of the variables is the search order. In a standard *nix environment these variables are not set, and there's no need to set them. On Windows it's recommended to set one of them.

SDCC_HOME Path, see section 2.2 " Install Paths".

SDCC_INCLUDE Path, see section 2.3 "Search Paths".

SDCC_LIB Path, see section 2.3 "Search Paths"..

There are some more environment variables recognized by SDCC, but these are solely used for debugging purposes. They can change or disappear very quickly, and will never be documented.

3.4 Storage Class Language Extensions

3.4.1 MCS51/DS390 Storage Class Language Extensions

In addition to the ANSI storage classes SDCC allows the following MCS51 specific storage classes:

3.4.1.1 data / near

This is the **default** storage class for the Small Memory model (*data* and *near* can be used synonymously). Variables declared with this storage class will be allocated in the directly addressable portion of the internal RAM of a 8051, e.g.:

```
data unsigned char test_data;
```

Writing 0x01 to this variable generates the assembly code:

```
75*00 01   mov  _test_data,#0x01
```

3.4.1.2 xdata / far

Variables declared with this storage class will be placed in the external RAM. This is the **default** storage class for the Large Memory model, e.g.:

```
xdata unsigned char test_xdata;
```

Writing 0x01 to this variable generates the assembly code:

```
90s00r00   mov  dptr,#_test_xdata
74 01      mov  a,#0x01
F0         movx @dptr,a
```

3.4.1.3 idata

Variables declared with this storage class will be allocated into the indirectly addressable portion of the internal ram of a 8051, e.g.:

```
idata unsigned char test_idata;
```

Writing 0x01 to this variable generates the assembly code:

```
78r00       mov  r0,#_test_idata
76 01       mov  @r0,#0x01
```

Please note, the first 128 byte of idata physically access the same RAM as the data memory. The original 8051 had 128 byte idata memory, nowadays most devices have 256 byte idata memory. The stack is located in idata memory.

3.4.1.4 pdata

Paged xdata access is currently not as straightforward as using the other addressing modes of a 8051. The following example writes 0x01 to the address pointed to. Please note, pdata access physically accesses xdata memory. The high byte of the address is determined by port P2 (or in case of some 8051 variants by a separate Special Function Register, see section 4.1).

```
pdata unsigned char *test_pdata_ptr;

void main()
{
    test_pdata_ptr = (pdata *)0xfe;
    *test_pdata_ptr = 1;
}
```

Generates the assembly code:

```
75*01 FE   mov  _test_pdata_ptr,#0xFE
78 FE      mov  r0,#0xFE
74 01      mov  a,#0x01
F2         movx @r0,a
```

Be extremely carefull if you use pdata together with the --xstack option.

3.4.1.5 code

'Variables' declared with this storage class will be placed in the code memory:

```
code unsigned char test_code;
```

Read access to this variable generates the assembly code:

```
90s00r6F   mov dptr,#_test_code
E4         clr a
93         movc a,@a+dptr
```

`char` indexed arrays of characters in code memory can be accessed efficiently:

```
code char test_array[] = {'c','h','e','a','p'};
```

Read access to this array using an 8-bit unsigned index generates the assembly code:

```
E5*00      mov a,_index
90s00r41   mov dptr,#_test_array
93         movc a,@a+dptr
```

3.4.1.6 bit

This is a data-type and a storage class specifier. When a variable is declared as a bit, it is allocated into the bit addressable memory of 8051, e.g.:

```
bit test_bit;
```

Writing 1 to this variable generates the assembly code:

```
D2*00       setb _test_bit
```

The bit addressable memory consists of 128 bits which are located from 0x20 to 0x2f in data memory.

Apart from this 8051 specific storage class most architectures support ANSI-C bitfields [1].

[1] Not really meant as examples, but nevertheless showing what bitfields are about: device/include/mc68hc908qy.h and support/regression/tests/bitfields.c

3.4.1.7 sfr / sbit

Like the bit keyword, *sfr / sbit* signifies both a data-type and storage class, they are used to describe the *s*pecial *f*unction *r*egisters and *s*pecial *bit* variables of a 8051, eg:

```
sfr at 0x80 P0;  /* special function register P0 at location 0x80 */
sbit at 0xd7 CY; /* CY (Carry Flag) */
```

Special function registers which are located on an address dividable by 8 are bit-addressable, an *sbit* addresses a specific bit within these sfr.

3.4.1.8 Pointers to MCS51/DS390 specific memory spaces

SDCC allows (via language extensions) pointers to explicitly point to any of the memory spaces of the 8051. In addition to the explicit pointers, the compiler uses (by default) generic pointers which can be used to point to any of the memory spaces.

Pointer declaration examples:

```
/* pointer physically in internal ram pointing to object in external ram */
xdata unsigned char * data p;

/* pointer physically in external ram pointing to object in internal ram */
data unsigned char * xdata p;

/* pointer physically in code rom pointing to data in xdata space */
xdata unsigned char * code p;

/* pointer physically in code space pointing to data in code space */
code unsigned char * code p;

/* the following is a generic pointer physically located in xdata space */
char * xdata p;
```

Well you get the idea.

All unqualified pointers are treated as 3-byte (4-byte for the ds390) *generic* pointers.

The highest order byte of the *generic* pointers contains the data space information. Assembler support routines are called whenever data is stored or retrieved using *generic* pointers. These are useful for developing reusable library routines. Explicitly specifying the pointer type will generate the most efficient code.

3.4.1.9 Notes on MCS51 memory layout

The 8051 family of microcontrollers have a minimum of 128 bytes of internal RAM memory which is structured as follows:

- Bytes 00-1F - 32 bytes to hold up to 4 banks of the registers R0 to R7,
- Bytes 20-2F - 16 bytes to hold 128 bit variables and,
- Bytes 30-7F - 80 bytes for general purpose use.

Additionally some members of the MCS51 family may have up to 128 bytes of additional, indirectly addressable, internal RAM memory (*idata*). Furthermore, some chips may have some built in external memory (*xdata*) which should not be confused with the internal, directly addressable RAM memory (*data*). Sometimes this built in *xdata* memory has to be activated before using it (you can probably find this information on the datasheet of the microcontroller your are using, see also section 3.11 Startup-Code).

Normally SDCC will only use the first bank of registers (register bank 0), but it is possible to specify that other banks of registers should be used in interrupt routines. By default, the compiler will place the stack after the last

byte of allocated memory for variables. For example, if the first 2 banks of registers are used, and only four bytes are used for *data* variables, it will position the base of the internal stack at address 20 (0x14). This implies that as the stack grows, it will use up the remaining register banks, and the 16 bytes used by the 128 bit variables, and 80 bytes for general purpose use. If any bit variables are used, the data variables will be placed after the byte holding the last bit variable. For example, if register banks 0 and 1 are used, and there are 9 bit variables (two bytes used), *data* variables will be placed starting at address 0x22. You can also use --data-loc to specify the start address of the *data* and --iram-size to specify the size of the total internal RAM (*data*+*idata*).

By default the 8051 linker will place the stack after the last byte of data variables. Option --stack-loc allows you to specify the start of the stack, i.e. you could start it after any data in the general purpose area. If your microcontroller has additional indirectly addressable internal RAM (*idata*) you can place the stack on it. You may also need to use --xdata-loc to set the start address of the external RAM (*xdata*) and --xram-size to specify its size. Same goes for the code memory, using --code-loc and --code-size. If in doubt, don't specify any options and see if the resulting memory layout is appropriate, then you can adjust it.

The linker generates two files with memory allocation information. The first, with extension .map shows all the variables and segments. The second with extension .mem shows the final memory layout. The linker will complain either if memory segments overlap, there is not enough memory, or there is not enough space for stack. If you get any linking warnings and/or errors related to stack or segments allocation, take a look at either the .map or .mem files to find out what the problem is. The .mem file may even suggest a solution to the problem.

3.4.2 Z80/Z180 Storage Class Language Extensions

3.4.2.1 sfr (in/out to 8-bit addresses)

The Z80 family has separate address spaces for memory and *i*nput/*o*utput memory. I/O memory is accessed with special instructions, e.g.:

```
sfr at 0x78 IoPort;  /* define a var in I/O space at 78h called IoPort */
```

Writing 0x01 to this variable generates the assembly code:

```
3E 01      ld a,#0x01
D3 78      out (_IoPort),a
```

3.4.2.2 banked sfr (in/out to 16-bit addresses)

The keyword *banked* is used to support 16 bit addresses in I/O memory e.g.:

```
sfr banked at 0x123 IoPort;
```

Writing 0x01 to this variable generates the assembly code:

```
01 23 01   ld bc,#_IoPort
3E 01      ld a,#0x01
ED 79      out (c),a
```

3.4.2.3 sfr (in0/out0 to 8 bit addresses on Z180/HD64180)

The compiler option --portmode=180 (80) and a compiler #pragma portmode=z180 (z80) is used to turn on (off) the Z180/HD64180 port addressing instructions `in0/out0` instead of `in/out`. If you include the file z180.h this will be set automatically.

3.4.3 HC08 Storage Class Language Extensions

3.4.3.1 data

The data storage class declares a variable that resides in the first 256 bytes of memory (the direct page). The HC08 is most efficient at accessing variables (especially pointers) stored here.

3.4.3.2 xdata

The xdata storage class declares a variable that can reside anywhere in memory. This is the default if no storage class is specified.

3.5 Absolute Addressing

Data items can be assigned an absolute address with the *at <address>* keyword, in addition to a storage class, e.g.:

```
xdata at 0x7ffe unsigned int chksum;
```

In the above example the variable chksum will located at 0x7ffe and 0x7fff of the external ram. The compiler does *not* reserve any space for variables declared in this way (they are implemented with an equate in the assembler). Thus it is left to the programmer to make sure there are no overlaps with other variables that are declared without the absolute address. The assembler listing file (.lst) and the linker output files (.rst) and (.map) are good places to look for such overlaps. Variables with an absolute address are *not* initialized.

In case of memory mapped I/O devices the keyword *volatile* should be used to tell the compiler that accesses might not be optimized away:

```
volatile xdata at 0x8000 unsigned char PORTA_8255;
```

For some architectures (mcs51) array accesses are more efficient if an (xdata/far) array starts at a block (256 byte) boundary (section 3.12.1 has an example).
Absolute addresses can be specified for variables in all storage classes, e.g.:

```
bit at 0x02 bvar;
```

The above example will allocate the variable at offset 0x02 in the bit-addressable space. There is no real advantage to assigning absolute addresses to variables in this manner, unless you want strict control over all the variables allocated. One possible use would be to write hardware portable code. For example, if you have a routine that uses one or more of the microcontroller I/O pins, and such pins are different for two different hardwares, you can declare the I/O pins in your routine using:

```
extern volatile bit SDI;
extern volatile bit SCLK;
extern volatile bit CPOL;

void DS1306_put(unsigned char value)
{
    unsigned char mask=0x80;

    while(mask)
    {
        SDI=(value & mask)?1:0;
        SCLK=!CPOL;
        SCLK=CPOL;
        mask/=2;
    }
}
```

Then, someplace in the code for the first hardware you would use

```
bit at 0x80 SDI;    /* I/O port 0, bit 0 */
bit at 0x81 SCLK;   /* I/O port 0, bit 1 */
bit CPOL;           /* This is a variable, let the linker allocate this one */
```

Similarly, for the second hardware you would use

```
bit at 0x83 SDI;     /* I/O port 0, bit 3 */
bit at 0x91 SCLK;    /* I/O port 1, bit 1 */
bit CPOL;            /* This is a variable, let the linker allocate this one */
```

and you can use the same hardware dependent routine without changes, as for example in a library. This is somehow similar to sbit, but only one absolute address has to be specified in the whole project.

3.6 Parameters & Local Variables

Automatic (local) variables and parameters to functions can either be placed on the stack or in data-space. The default action of the compiler is to place these variables in the internal RAM (for small model) or external RAM (for large model). This in fact makes them similar to *static* so by default functions are non-reentrant.

They can be placed on the stack either by using the *--stack-auto* option or by using the *reentrant* keyword in the function declaration, e.g.:

```
unsigned char foo(char i) reentrant
{
    ...
}
```

Since stack space on 8051 is limited, the *reentrant* keyword or the *--stack-auto* option should be used sparingly. Note that the reentrant keyword just means that the parameters & local variables will be allocated to the stack, it *does not* mean that the function is register bank independent.

Local variables can be assigned storage classes and absolute addresses, e.g.:

```
unsigned char foo()
{
    xdata unsigned char i;
    bit bvar;
    data at 0x31 unsigned char j;
    ...
}
```

In the above example the variable *i* will be allocated in the external ram, *bvar* in bit addressable space and *j* in internal ram. When compiled with *--stack-auto* or when a function is declared as *reentrant* this should only be done for static variables.

Parameters however are not allowed any storage class, (storage classes for parameters will be ignored), their allocation is governed by the memory model in use, and the reentrancy options.

3.7 Overlaying

For non-reentrant functions SDCC will try to reduce internal ram space usage by overlaying parameters and local variables of a function (if possible). Parameters and local variables of a function will be allocated to an overlayable segment if the function has *no other function calls and the function is non-reentrant and the memory model is small.* If an explicit storage class is specified for a local variable, it will NOT be overlayed.

Note that the compiler (not the linkage editor) makes the decision for overlaying the data items. Functions that are called from an interrupt service routine should be preceded by a #pragma nooverlay if they are not reentrant.

Also note that the compiler does not do any processing of inline assembler code, so the compiler might incorrectly assign local variables and parameters of a function into the overlay segment if the inline assembler code calls other c-functions that might use the overlay. In that case the #pragma nooverlay should be used.

Parameters and local variables of functions that contain 16 or 32 bit multiplication or division will NOT be overlayed since these are implemented using external functions, e.g.:

```
#pragma save
#pragma nooverlay
void set_error(unsigned char errcd)
{
    P3 = errcd;
}
#pragma restore

void some_isr () interrupt 2
{
    ...
    set_error(10);
    ...
}
```

In the above example the parameter *errcd* for the function *set_error* would be assigned to the overlayable segment if the #pragma nooverlay was not present, this could cause unpredictable runtime behavior when called from an interrupt service routine. The #pragma nooverlay ensures that the parameters and local variables for the function are NOT overlayed.

3.8 Interrupt Service Routines

3.8.1 General Information

SDCC allows *i*nterrupt *s*ervice *r*outines to be coded in C, with some extended keywords.

```
void timer_isr (void) interrupt 1 using 1
{
    ...
}
```

The optional number following the *interrupt* keyword is the interrupt number this routine will service. When present, the compiler will insert a call to this routine in the interrupt vector table for the interrupt number specified. If you have multiple source files in your project, interrupt service routines can be present in any of them, but a prototype of the isr MUST be present or included in the file that contains the function *main*. The *using* keyword can be used to tell the compiler to use the specified register bank (8051 specific) when generating code for this function.

Interrupt service routines open the door for some very interesting bugs:

If the interrupt service routines changes variables which are accessed by other functions these variables should be declared *volatile*.

If the access to these variables is not *atomic* (i.e. the processor needs more than one instruction for the access and could be interrupted while accessing the variable) the interrupt must disabled during the access to avoid inconsistent data. Access to 16 or 32 bit variables is obviously not atomic on 8 bit CPUs and should be protected by disabling interrupts. You're not automatically on the safe side if you use 8 bit variables though. We need an example here: f.e. on the 8051 the harmless looking "`flags |= 0x80;`" is not atomic if `flags` resides in xdata. Setting "`flags |= 0x40;`" from within an interrupt routine might get lost if the interrupt occurs at the wrong time. "`counter += 8;`" is not atomic on the 8051 even if `counter` is located in data memory. Bugs like these are hard to reproduce and can cause a lot of trouble.

A special note here, int (16 bit) and long (32 bit) integer division, multiplication & modulus and floating-point operations are implemented using external support routines developed in ANSI-C. If an interrupt service routine needs to do any of these operations then the support routines (as mentioned in a following section) will have to be recompiled using the *--stack-auto* option and the source file will need to be compiled using the *--int-long-reent* compiler option.

Calling other functions from an interrupt service routine is not recommended, avoid it if possible. Note that when some function is called from an interrupt service routine it should be preceded by a #pragma nooverlay if it is not reentrant. Furthermore nonreentrant functions should not be called from the main program while the interrupt service routine might be active.

Also see section 3.7 about Overlaying and section 3.10 about Functions using private register banks.

3.8.2 MCS51/DS390 Interrupt Service Routines

Interrupt numbers and the corresponding address & descriptions for the Standard 8051/8052 are listed below. SDCC will automatically adjust the interrupt vector table to the maximum interrupt number specified.

Interrupt #	Description	Vector Address
0	External 0	0x0003
1	Timer 0	0x000B
2	External 1	0x0013
3	Timer 1	0x001B
4	Serial	0x0023
5	Timer 2 (8052)	0x002B

If the interrupt service routine is defined without *using* a register bank or with register bank 0 (*using* 0), the compiler will save the registers used by itself on the stack upon entry and restore them at exit, however if such an interrupt service routine calls another function then the entire register bank will be saved on the stack. This scheme may be advantageous for small interrupt service routines which have low register usage.

If the interrupt service routine is defined to be using a specific register bank then only *a, b, dptr* & psw are saved and restored, if such an interrupt service routine calls another function (using another register bank) then the entire register bank of the called function will be saved on the stack. This scheme is recommended for larger interrupt service routines.

3.8.3 HC08 Interrupt Service Routines

Since the number of interrupts available is chip specific and the interrupt vector table always ends at the last byte of memory, the interrupt numbers corresponds to the interrupt vectors in reverse order of address. For example, interrupt 1 will use the interrupt vector at 0xfffc, interrupt 2 will use the interrupt vector at 0xfffa, and so on. However, interrupt 0 (the reset vector at 0xfffe) is not redefinable in this way; instead see section 3.11 for details on customizing startup.

3.9 Enabling and Disabling Interrupts

3.9.1 Critical Functions and Critical Statements

A special keyword may be associated with a block or a function declaring it as *critical*. SDCC will generate code to disable all interrupts upon entry to a critical function and restore the interrupt enable to the previous state before returning. Nesting critical functions will need one additional byte on the stack for each call.

```
int foo () critical
{
    ...
    ...
}
```

The critical attribute maybe used with other attributes like *reentrant.*
The keyword *critical* may also be used to disable interrupts more locally:

```
critical{ i++; }
```

More than one statement could have been included in the block.

3.9.2 Enabling and Disabling Interrupts directly

Interrupts can also be disabled and enabled directly (8051):

```
EA = 0;            or:          EA_SAVE = EA;
...                             EA = 0;
EA = 1;                         ...
                                EA = EA_SAVE;
```

On other architectures which have seperate opcodes for enabling and disabling interrupts you might want to make use of defines with inline assembly (HC08):

```
#define CLI _asm  cli  _endasm;
#define SEI _asm  sei  _endasm;
...
```

Note: it is sometimes sufficient to disable only a specific interrupt source like f.e. a timer or serial interrupt by manipulating an *interrupt mask* register.

Usually the time during which interrupts are disabled should be kept as short as possible. This minimizes both *interrupt latency* (the time between the occurrence of the interrupt and the execution of the first code in the interrupt routine) and *interrupt jitter* (the difference between the shortest and the longest interrupt latency). These really are something different, f.e. a serial interrupt has to be served before its buffer overruns so it cares for the maximum interrupt latency, whereas it does not care about jitter. On a loudspeaker driven via a digital to analog converter which is fed by an interrupt a latency of a few milliseconds might be tolerable, whereas a much smaller jitter will be very audible.

You can reenable interrupts within an interrupt routine and on some architectures you can make use of two (or more) levels of *interrupt priorities*. On some architectures which don't support interrupt priorities these can be implemented by manipulating the interrupt mask and reenabling interrupts within the interrupt routine. Don't add complexity unless you have to.

3.10 Functions using private register banks (mcs51/ds390)

Some architectures have support for quickly changing register sets. SDCC supports this feature with the *using* attribute (which tells the compiler to use a register bank other than the default bank zero). It should only be applied to *interrupt* functions (see footnote below). This will in most circumstances make the generated ISR code more efficient since it will not have to save registers on the stack.

The *using* attribute will have no effect on the generated code for a *non-interrupt* function (but may occasionally be useful anyway[2]).

(pending: I don't think this has been done yet)

An *interrupt* function using a non-zero bank will assume that it can trash that register bank, and will not save it. Since high-priority interrupts can interrupt low-priority ones on the 8051 and friends, this means that if a high-priority ISR *using* a particular bank occurs while processing a low-priority ISR *using* the same bank, terrible and bad things can happen. To prevent this, no single register bank should be *used* by both a high priority and a low priority ISR. This is probably most easily done by having all high priority ISRs use one bank and all low priority ISRs use another. If you have an ISR which can change priority at runtime, you're on your own: I suggest using the default bank zero and taking the small performance hit.

It is most efficient if your ISR calls no other functions. If your ISR must call other functions, it is most efficient if those functions use the same bank as the ISR (see note 1 below); the next best is if the called functions use bank zero. It is very inefficient to call a function using a different, non-zero bank from an ISR.

[2] possible exception: if a function is called ONLY from 'interrupt' functions using a particular bank, it can be declared with the same 'using' attribute as the calling 'interrupt' functions. For instance, if you have several ISRs using bank one, and all of them call memcpy(), it might make sense to create a specialized version of memcpy() 'using 1', since this would prevent the ISR from having to save bank zero to the stack on entry and switch to bank zero before calling the function

3.11 Startup Code

3.11.1 MCS51/DS390 Startup Code

The compiler inserts a call to the C routine *_sdcc_external_startup()* at the start of the CODE area. This routine is in the runtime library. By default this routine returns 0, if this routine returns a non-zero value, the static & global variable initialization will be skipped and the function main will be invoked. Otherwise static & global variables will be initialized before the function main is invoked. You could add a *_sdcc_external_startup()* routine to your program to override the default if you need to setup hardware or perform some other critical operation prior to static & global variable initialization. On some mcs51 variants xdata has to be explicitly enabled before it can be accessed, this is the place to do it. See also the compiler option *--no-xinit-opt* and section 4.1 about MCS51-variants.

3.11.2 HC08 Startup Code

The HC08 startup code follows the same scheme as the MCS51 startup code.

3.11.3 Z80 Startup Code

On the Z80 the startup code is inserted by linking with crt0.o which is generated from sdcc/device/lib/z80/crt0.s. If you need a different startup code you can use the compiler option *--no-std-crt0* and provide your own crt0.o.

3.12 Inline Assembler Code

3.12.1 A Step by Step Introduction

Starting from a small snippet of c-code this example shows for the MCS51 how to use inline assembly, access variables, a function parameter and an array in xdata memory. The example uses an MCS51 here but is easily adapted for other architectures. This is a buffer routine which should be optimized:

```
unsigned char far at 0x7f00 buf[0x100];
unsigned char head,tail;

void to_buffer( unsigned char c )
{
    if( head != tail-1 )
        buf[ head++ ] = c;
}
```

If the code snippet (assume it is saved in buffer.c) is compiled with SDCC then a corresponding buffer.asm file is generated. We define a new function `to_buffer_asm()` in file buffer.c in which we cut and paste the generated code, removing unwanted comments and some ':'. Then add "_asm" and "_endasm;" to the beginning and the end of the function body:

```
/* With a cut and paste from the .asm file, we have something to start with.
   The function is not yet OK! (registers aren't saved) */
void to_buffer_asm( unsigned char c )
{
    _asm
    mov  r2,dpl
;buffer.c if( head != tail-1 )
    mov  a,_tail
    dec  a
    mov  r3,a
    mov  a,_head
    cjne a,ar3,00106$
    ret
00106$:
;buffer.c buf[ head++ ] = c; /* access to a 256 byte aligned array */
```

```
    mov  r3,_head
    inc  _head
    mov  dpl,r3
    mov  dph,#(_buf >> 8)
    mov  a,r2
    movx @dptr,a
00103$:
    ret
    _endasm;
}
```

The new file buffer.c should compile with only one warning about the unreferenced function argument 'c'. Now we hand-optimize the assembly code and insert an #define USE_ASSEMBLY (1) and finally have:

```
unsigned char far at 0x7f00 buf[0x100];
unsigned char head,tail;
#define USE_ASSEMBLY (1)

#if !USE_ASSEMBLY

void to_buffer( unsigned char c )
{
    if( head != tail-1 )
        buf[ head++ ] = c;
}

#else

void to_buffer( unsigned char c )
{
    c; // to avoid warning:  unreferenced function argument
    _asm
        ; save used registers here.
        ; If we were still using r2,r3 we would have to push them here.
; if( head != tail-1 )
        mov  a,_tail
        dec  a
        xrl  a,_head
        ; we could do an ANL a,#0x0f here to use a smaller buffer (see below)
        jz   t_b_end$
        ;
; buf[ head++ ] = c;
        mov  a,dpl        ; dpl holds lower byte of function argument
        mov  dpl,_head    ; buf is 0x100 byte aligned so head can be used directly
        mov  dph,#(_buf>>8)
        movx @dptr,a
        inc  _head
        ; we could do an ANL _head,#0x0f here to use a smaller buffer (see above)
t_b_end$:
        ; restore used registers here
    _endasm;
}
#endif
```

The inline assembler code can contain any valid code understood by the assembler, this includes any assembler directives and comment lines[3]. The compiler does not do any validation of the code within the `_asm ... _endasm;`

[3]The assembler does not like some characters like ':' or "' in comments. You'll find an 100+ pages assembler manual in sdcc/as/doc/asxhtm.html

keyword pair. Specifically it will not know which registers are used and thus register pushing/popping has to be done manually.

It is recommended that each assembly instruction (including labels) be placed in a separate line (as the example shows). When the *--peep-asm* command line option is used, the inline assembler code will be passed through the peephole optimizer. There are only a few (if any) cases where this option makes sense, it might cause some unexpected changes in the inline assembler code. Please go through the peephole optimizer rules defined in file *SDCCpeeph.def* before using this option.

3.12.2 Naked Functions

A special keyword may be associated with a function declaring it as *_naked*. The *_naked* function modifier attribute prevents the compiler from generating prologue and epilogue code for that function. This means that the user is entirely responsible for such things as saving any registers that may need to be preserved, selecting the proper register bank, generating the *return* instruction at the end, etc. Practically, this means that the contents of the function must be written in inline assembler. This is particularly useful for interrupt functions, which can have a large (and often unnecessary) prologue/epilogue. For example, compare the code generated by these two functions:

```
volatile data unsigned char counter;

void simpleInterrupt(void) interrupt 1
{
    counter++;
}

void nakedInterrupt(void) interrupt 2 _naked
{
    _asm
      inc     _counter ; does not change flags, no need to save psw
      reti    ; MUST explicitly include ret or reti in _naked function.
    _endasm;
}
```

For an 8051 target, the generated simpleInterrupt looks like:

```
_simpleInterrupt:
    push    acc
    push    b
    push    dpl
    push    dph
    push    psw
    mov     psw,#0x00
    inc     _counter
    pop     psw
    pop     dph
    pop     dpl
    pop     b
    pop     acc
    reti
```

whereas nakedInterrupt looks like:

```
_nakedInterrupt:
    inc    _counter ; does not change flags, no need to save psw
    reti            ; MUST explicitly include ret or reti in _naked function
```

The related directive #pragma exclude allows a more fine grained control over pushing & popping the registers.

While there is nothing preventing you from writing C code inside a `_naked` function, there are many ways to shoot yourself in the foot doing this, and it is recommended that you stick to inline assembler.

3.12.3 Use of Labels within Inline Assembler

SDCC allows the use of in-line assembler with a few restrictions regarding labels. In older versions of the compiler all labels defined within inline assembler code *had to be* of the form *nnnnn$* where nnnn is a number less than 100 (which implies a limit of utmost 100 inline assembler labels *per function*).

```
_asm
    mov     b,#10
00001$:
    djnz    b,00001$
_endasm ;
```

Inline assembler code cannot reference any C-Labels, however it can reference labels defined by the inline assembler, e.g.:

```
foo() {
    /* some c code */
    _asm
      ; some assembler code
      ljmp $0003
    _endasm;
    /* some more c code */
clabel:  /* inline assembler cannot reference this label */
    _asm
    $0003:  ;label (can be referenced by inline assembler only)
    _endasm ;
    /* some more c code */
}
```

In other words inline assembly code can access labels defined in inline assembly within the scope of the function. The same goes the other way, i.e. labels defines in inline assembly can not be accessed by C statements.

3.13 Interfacing with Assembler Code

3.13.1 Global Registers used for Parameter Passing

The compiler always uses the global registers *DPL, DPH, B* and *ACC* to pass the first parameter to a routine. The second parameter onwards is either allocated on the stack (for reentrant routines or if --stack-auto is used) or in data / xdata memory (depending on the memory model).

3.13.2 Assembler Routine (non-reentrant)

In the following example the function c_func calls an assembler routine asm_func, which takes two parameters.

```
extern int asm_func(unsigned char, unsigned char);

int c_func (unsigned char i, unsigned char j)
{
    return asm_func(i,j);
}

int main()
{
    return c_func(10,9);
}
```

The corresponding assembler function is:

```
.globl _asm_func_PARM_2
        .globl _asm_func
        .area OSEG
_asm_func_PARM_2:
        .ds 1
        .area CSEG
_asm_func:
        mov    a,dpl
        add    a,_asm_func_PARM_2
        mov    dpl,a
        mov    dph,#0x00
        ret
```

Note here that the return values are placed in 'dpl' - One byte return value, 'dpl' LSB & 'dph' MSB for two byte values. 'dpl', 'dph' and 'b' for three byte values (generic pointers) and 'dpl','dph','b' & 'acc' for four byte values.

The parameter naming convention is _<function_name>_PARM_<n>, where n is the parameter number starting from 1, and counting from the left. The first parameter is passed in "dpl" for a one byte parameter, "dptr" for two bytes, "b,dptr" for three bytes and "acc,b,dptr" for a four bytes parameter. The variable name for the second parameter will be _<function_name>_PARM_2.

Assemble the assembler routine with the following command:

asx8051 -losg asmfunc.asm

Then compile and link the assembler routine to the C source file with the following command:

sdcc cfunc.c asmfunc.rel

3.13.3 Assembler Routine (reentrant)

In this case the second parameter onwards will be passed on the stack, the parameters are pushed from right to left i.e. after the call the leftmost parameter will be on the top of the stack. Here is an example:

```
extern int asm_func(unsigned char, unsigned char);

int c_func (unsigned char i, unsigned char j) reentrant
{
    return asm_func(i,j);
}

int main()
{
    return c_func(10,9);
}
```

The corresponding assembler routine is:

```
.globl _asm_func
_asm_func:
    push _bp
    mov _bp,sp
    mov r2,dpl
    mov a,_bp
    add a,#0xfd
    mov r0,a
    add a,#0xfc ;?
    mov r1,a
    mov a,@r0
```

```
add a,r2 ;?
mov dpl,a
mov dph,#0x00
mov sp,_bp
pop _bp
ret
```

The compiling and linking procedure remains the same, however note the extra entry & exit linkage required for the assembler code, _bp is the stack frame pointer and is used to compute the offset into the stack for parameters and local variables.

3.14 int (16 bit) and long (32 bit) Support

For signed & unsigned int (16 bit) and long (32 bit) variables, division, multiplication and modulus operations are implemented by support routines. These support routines are all developed in ANSI-C to facilitate porting to other MCUs, although some model specific assembler optimizations are used. The following files contain the described routines, all of them can be found in <installdir>/share/sdcc/lib.

Function	Description
_mulint.c	16 bit multiplication
_divsint.c	signed 16 bit division (calls _divuint)
_divuint.c	unsigned 16 bit division
_modsint.c	signed 16 bit modulus (calls _moduint)
_moduint.c	unsigned 16 bit modulus
_mullong.c	32 bit multiplication
_divslong.c	signed 32 division (calls _divulong)
_divulong.c	unsigned 32 division
_modslong.c	signed 32 bit modulus (calls _modulong)
_modulong.c	unsigned 32 bit modulus

Since they are compiled as *non-reentrant*, interrupt service routines should not do any of the above operations. If this is unavoidable then the above routines will need to be compiled with the *--stack-auto* option, after which the source program will have to be compiled with *--int-long-reent* option. Notice that you don't have to call these routines directly. The compiler will use them automatically every time an integer operation is required.

3.15 Floating Point Support

SDCC supports IEEE (single precision 4 bytes) floating point numbers.The floating point support routines are derived from gcc's floatlib.c and consist of the following routines:

Function	Description
_fsadd.c	add floating point numbers
_fssub.c	subtract floating point numbers
_fsdiv.c	divide floating point numbers
_fsmul.c	multiply floating point numbers
_fs2uchar.c	convert floating point to unsigned char
_fs2char.c	convert floating point to signed char
_fs2uint.c	convert floating point to unsigned int
_fs2int.c	convert floating point to signed int
_fs2ulong.c	convert floating point to unsigned long
_fs2long.c	convert floating point to signed long
_uchar2fs.c	convert unsigned char to floating point
_char2fs.c	convert char to floating point number
_uint2fs.c	convert unsigned int to floating point
_int2fs.c	convert int to floating point numbers
_ulong2fs.c	convert unsigned long to floating point number
_long2fs.c	convert long to floating point number

These support routines are developed in ANSI-C so there is room for space and speed improvement. Note if all these routines are used simultaneously the data space might overflow. For serious floating point usage it is recommended that the large model be used. Also notice that you don't have to call this routines directly. The compiler will use them automatically every time a floating point operation is required.

3.16 Library Routines

<pending: this is messy and incomplete - a little more information is in sdcc/doc/libdoc.txt >

3.16.1 Compiler support routines (_gptrget, _mulint etc.)

3.16.2 Stdclib functions (puts, printf, strcat etc.)

3.16.2.1 <stdio.h>

As usual on embedded systems you have to provide your own `getchar()` and `putchar()` routines. SDCC does not know whether the system connects to a serial line with or without handshake, LCD, keyboard or other device. You'll find examples for serial routines f.e. in sdcc/device/lib.

If you're short on memory you might want to use `printf_small()` *instead* of `printf()`. For the mcs51 there is an assembly version `printf_fast()` which should fit the requirements of many embedded systems (by unsetting #defines it can be customized to *not* support long variables and field widths).

3.16.3 Math functions (sin, pow, sqrt etc.)

3.16.4 Other libraries

Libraries included in SDCC should have a license at least as liberal as the GNU Lesser General Public License *LGPL*.

If you have ported some library or want to share experience about some code which f.e. falls into any of these categories Busses (I^2C, CAN, Ethernet, Profibus, Modbus, USB, SPI, JTAG ...), Media (IDE, Memory cards, eeprom, flash...), En-/Decryption, Remote debugging, Realtime kernel, Keyboard, LCD, RTC, FPGA, PID then the sdcc-user mailing list `http://sourceforge.net/mail/?group_id=599` would certainly like to hear about it. Programmers coding for embedded systems are not especially famous for being enthusiastic, so don't expect a big hurray but as the mailing list is searchable these references are very valuable. Let's help to create a climate where information is shared.

3.17 Memory Models

3.17.1 MCS51 Memory Models

3.17.1.1 Small and Large

SDCC allows two memory models for MCS51 code, *small* and *large*. Modules compiled with different memory models should *never* be combined together or the results would be unpredictable. The library routines supplied with the compiler are compiled as both small and large. The compiled library modules are contained in separate directories as small and large so that you can link to either set.

When the large model is used all variables declared without a storage class will be allocated into the external ram, this includes all parameters and local variables (for non-reentrant functions). When the small model is used variables without storage class are allocated in the internal ram.

Judicious usage of the processor specific storage classes and the 'reentrant' function type will yield much more efficient code, than using the large model. Several optimizations are disabled when the program is compiled using the large model, it is therefore recommended that the small model be used unless absolutely required.

3.17.1.2 External Stack

Attention: this option wasn't maintained for a long time and is quite buggy. Small programs might work. You've been warned!

The external stack (--xstack option) is located in pdata memory (usually at the start of the external ram segment) and is 256 bytes in size. When --xstack option is used to compile the program, the parameters and local variables of all reentrant functions are allocated in this area. This option is provided for programs with large stack space requirements. When used with the --stack-auto option, all parameters and local variables are allocated on the external stack (note: support libraries will need to be recompiled with the same options).

The compiler outputs the higher order address byte of the external ram segment into port P2 (see also section 4.1), therefore when using the External Stack option, this port *may not* be used by the application program.

3.17.2 DS390 Memory Model

The only model supported is Flat 24. This generates code for the 24 bit contiguous addressing mode of the Dallas DS80C390 part. In this mode, up to four meg of external RAM or code space can be directly addressed. See the data sheets at www.dalsemi.com for further information on this part.

Note that the compiler does not generate any code to place the processor into 24 bitmode (although *tinibios* in the ds390 libraries will do that for you). If you don't use *tinibios*, the boot loader or similar code must ensure that the processor is in 24 bit contiguous addressing mode before calling the SDCC startup code.

Like the *--model-large* option, variables will by default be placed into the XDATA segment.

Segments may be placed anywhere in the 4 meg address space using the usual --*-loc options. Note that if any segments are located above 64K, the -r flag must be passed to the linker to generate the proper segment relocations, and the Intel HEX output format must be used. The -r flag can be passed to the linker by using the option *-Wl-r* on the SDCC command line. However, currently the linker can not handle code segments > 64k.

3.18 Pragmas

SDCC supports the following #pragma directives:

- save - this will save all current options to the save/restore stack. See restore.
- restore - will restore saved options from the last save. saves & restores can be nested. SDCC uses a save/restore stack: save pushes current options to the stack, restore pulls current options from the stack. See save.
- nogcse - will stop global common subexpression elimination.

- noinduction - will stop loop induction optimizations.
- nojtbound - will not generate code for boundary value checking, when switch statements are turned into jump-tables (dangerous). For more details see section 8.1.7.
- nooverlay - the compiler will not overlay the parameters and local variables of a function.
- less_pedantic - the compiler will not warn you anymore for obvious mistakes, you'r on your own now ;-(
- noloopreverse - Will not do loop reversal optimization
- exclude none | {acc[,b[,dpl[,dph]]] - The exclude pragma disables generation of pair of push/pop instruction in ISR function (using interrupt keyword). The directive should be placed immediately before the ISR function definition and it affects ALL ISR functions following it. To enable the normal register saving for ISR functions use #pragma exclude none.
- noiv - Do not generate interrupt vector table entries for all ISR functions defined after the pragma. This is useful in cases where the interrupt vector table must be defined manually, or when there is a secondary, manually defined interrupt vector table (e.g. for the autovector feature of the Cypress EZ-USB FX2). More elegantly this can be achieved by obmitting the optional interrupt number after the interrupt keyword, see section 3.8 about interrupts.
- callee_saves function1[,function2[,function3...]] - The compiler by default uses a caller saves convention for register saving across function calls, however this can cause unnecessary register pushing & popping when calling small functions from larger functions. This option can be used to switch off the register saving convention for the function names specified. The compiler will not save registers when calling these functions, extra code need to be manually inserted at the entry & exit for these functions to save & restore the registers used by these functions, this can SUBSTANTIALLY reduce code & improve run time performance of the generated code. In the future the compiler (with inter procedural analysis) may be able to determine the appropriate scheme to use for each function call. If --callee-saves command line option is used, the function names specified in #pragma callee_saves is appended to the list of functions specified in the command line.

SDCPP supports the following #pragma directives:

- preproc_asm (+ | -) - switch _asm _endasm block preprocessing on / off. Default is on.

The pragma's are intended to be used to turn-on or off certain optimizations which might cause the compiler to generate extra stack / data space to store compiler generated temporary variables. This usually happens in large functions. Pragma directives should be used as shown in the following example, they are used to control options & optimizations for a given function; pragmas should be placed before and/or after a function, placing pragma's inside a function body could have unpredictable results.

```
#pragma save        /* save the current settings */
#pragma nogcse      /* turnoff global subexpression elimination */
#pragma noinduction /* turn off induction optimizations */
int foo ()
{
    ...
    /* large code */
    ...
}
#pragma restore /* turn the optimizations back on */
```

The compiler will generate a warning message when extra space is allocated. It is strongly recommended that the save and restore pragma's be used when changing options for a function.

3.19 Defines Created by the Compiler

The compiler creates the following #defines:

#define	Description
SDCC	this Symbol is always defined
SDCC_mcs51 or SDCC_ds390 or SDCC_z80, etc	depending on the model used (e.g.: -mds390
__mcs51, __ds390, __hc08, __z80, etc	depending on the model used (e.g. -mz80)
SDCC_STACK_AUTO	when *--stack-auto* option is used
SDCC_MODEL_SMALL	when *--model-small* is used
SDCC_MODEL_LARGE	when *--model-large* is used
SDCC_USE_XSTACK	when *--xstack* option is used
SDCC_STACK_TENBIT	when *-mds390* is used
SDCC_MODEL_FLAT24	when *-mds390* is used

Chapter 6

TIPS

Here are a few guidelines that will help the compiler generate more efficient code, some of the tips are specific to this compiler others are generally good programming practice.

- Use the smallest data type to represent your data-value. If it is known in advance that the value is going to be less than 256 then use an 'unsigned char' instead of a 'short' or 'int'. Please note, that ANSI C requires both signed and unsigned chars to be promoted to 'signed int' before doing any operation. This promotion can be omitted, if the result is the same. The effect of the promotion rules together with the sign-extension is often supprising:

  ```
  unsigned char uc = 0xfe;
  if (uc * uc < 0) /* this is true!  */
  {
      ....
  }
  ```

  `uc * uc` is evaluated as `(int) uc * (int) uc = (int) 0xfe * (int) 0xfe = (int) 0xfc04 = -1024`.
  Another one:

  ```
  (unsigned char) -12 / (signed char) -3 = ...
  ```

  No, the result is not 4:

  ```
  (int) (unsigned char) -12 / (int) (signed char) -3 =
  (int) (unsigned char) 0xf4 / (int) (signed char) 0xfd =
  (int) 0x00f4 / (int) 0xfffd =
  (int) 0x00f4 / (int) 0xfffd =
  (int) 244 / (int) -3 =
  (int) -81 = (int) 0xffaf;
  ```

  Don't complain, that gcc gives you a different result. gcc uses 32 bit ints, while SDCC uses 16 bit ints. Therefore the results are different.
  From "comp.lang.c FAQ":

  > If well-defined overflow characteristics are important and negative values are not, or if you want to steer clear of sign-extension problems when manipulating bits or bytes, use one of the corresponding unsigned types. (Beware when mixing signed and unsigned values in expressions, though.) Although character types (especially unsigned char) can be used as "tiny" integers, doing so is sometimes more trouble than it's worth, due to unpredictable sign extension and increased code size.

- Use unsigned when it is known in advance that the value is not going to be negative. This helps especially if you are doing division or multiplication, bit-shifting or are using an array index.

- NEVER jump into a LOOP.
- Declare the variables to be local whenever possible, especially loop control variables (induction).
- Since the compiler does not always do implicit integral promotion, the programmer should do an explicit cast when integral promotion is required.
- Reducing the size of division, multiplication & modulus operations can reduce code size substantially. Take the following code for example.

```
foobar(unsigned int p1, unsigned char ch)
{
    unsigned char ch1 = p1 % ch ;
    ....
}
```

For the modulus operation the variable ch will be promoted to unsigned int first then the modulus operation will be performed (this will lead to a call to support routine _moduint()), and the result will be casted to a char. If the code is changed to

```
foobar(unsigned int p1, unsigned char ch)
{
    unsigned char ch1 = (unsigned char)p1 % ch ;
    ....
}
```

It would substantially reduce the code generated (future versions of the compiler will be smart enough to detect such optimization opportunities).

- Have a look at the assembly listing to get a ”feeling” for the code generation.

6.1 Tools included in the distribution

Name	Purpose	Directory
uCsim	Simulator for various architectures	sdcc/sim/ucsim
keil2sdcc.pl	header file conversion	sdcc/support/scripts
mh2h.c	header file conversion	sdcc/support/scripts
as-gbz80	Assembler	sdcc/bin
as-z80	Assembler	sdcc/bin
asx8051	Assembler	sdcc/bin
sdcdb	Simulator	sdcc/bin
aslink	Linker	sdcc/bin
link-z80	Linker	sdcc/bin
link-gbz80	Linker	sdcc/bin
packihx	ihx packer	sdcc/bin

6.2 Documentation included in the distribution

Subject / Title	Where to get / filename
SDCC Compiler User Guide	You’re reading it right now
Changelog of SDCC	sdcc/Changelog
ASXXXX Assemblers and ASLINK Relocating Linker	sdcc/as/doc/asxhtm.html
SDCC regression test	sdcc/doc/test_suite_spec.pdf
Various notes	sdcc/doc/*
Notes on debugging with sdcdb	sdcc/debugger/README
Software simulator for microcontrollers	sdcc/sim/ucsim/doc/index.html
Temporary notes on the pic16 port	sdcc/src/pic16/NOTES
SDCC internal documentation (debugging file format)	sdcc/doc/cdbfileformat.pdf

6.3 Related open source tools

Name	Purpose	Where to get
gpsim	PIC simulator	http://www.dattalo.com/gnupic/gpsim.html
gputils	GNU PIC utilities	http://gputils.sourceforge.net/
flP5	PIC programmer	http://digilander.libero.it/fbradasc/FLP5.html
indent	Formats C source - Master of the white spaces	http://home.hccnet.nl/d.ingamells/beautify.html
srecord	Object file conversion, checksumming, ...	http://srecord.sourceforge.net/
objdump	Object file conversion, ...	Part of binutils (should be there anyway)
doxygen	Source code documentation system	http://www.doxygen.org
kdevelop	IDE (has anyone tried integrating SDCC & sdcdb? Unix only)	http://www.kdevelop.org
splint	Statically checks c sources	http://www.splint.org
ddd	Debugger, serves nicely as GUI to sdcdb (Unix only)	http://www.gnu.org/software/ddd/

6.4 Related documentation / recommended reading

Name	Subject / Title	Where to get
c-refcard.pdf	C Reference Card, 2 pages	http://www.refcards.com/about/c.html
c-faq	C-FAQ-list	http://www.eskimo.com/~scs/C-faq/top.html
	Latest datasheet of the target CPU	vendor
	Revision history of datasheet	vendor
S. S. Muchnick	Advanced Compiler Design and Implementation	bookstore (very dedicated, probably read other books first)

6.5 Some Questions

Some questions answered, some pointers given - it might be time to in turn ask *you* some questions:

- can you solve your project with the selected microcontroller? Would you find out early or rather late that your target is too small/slow/whatever? Can you switch to a slightly better device if it doesn't fit?
- should you solve the problem with an 8 bit CPU? Or would a 16/32 bit CPU and/or another programming language be more adequate? Would an operating system on the target device help?
- if you solved the problem, will the marketing department be happy?
- if the marketing department is happy, will customers be happy?
- if you're the project manager, marketing department and maybe even the customer in one person, have you tried to see the project from the outside?
- is the project done if you think it is done? Or is just that other interface/protocol/feature/configuration/option missing? How about website, manual(s), internationali(zls)ation, packaging, labels, 2nd source for components, electromagnetic compatability/interference, documentation for production, production test software, update mechanism, patent issues?
- is your project adequately positioned in that magic triangle: fame, fortune, fun?

Maybe not all answers to these questions are known and some answers may even be *no*, nevertheless knowing these questions may help you to avoid burnout[1]. Chances are you didn't want to hear some of them...

[1] burnout is bad for electronic devices, programmers and motorcycle tyres

Chapter 8

SDCC Technical Data

8.1 Optimizations

SDCC performs a host of standard optimizations in addition to some MCU specific optimizations.

8.1.1 Sub-expression Elimination

The compiler does local and *g*lobal *c*ommon *s*ubexpression *e*limination, e.g.:

```
i = x + y + 1;
j = x + y;
```

will be translated to

```
iTemp = x + y;
i = iTemp + 1;
j = iTemp;
```

Some subexpressions are not as obvious as the above example, e.g.:

```
a->b[i].c = 10;
a->b[i].d = 11;
```

In this case the address arithmetic a->b[i] will be computed only once; the equivalent code in C would be.

```
iTemp = a->b[i];
iTemp.c = 10;
iTemp.d = 11;
```

The compiler will try to keep these temporary variables in registers.

8.1.2 Dead-Code Elimination

```
int global;

void f () {
  int i;
  i = 1;      /* dead store */
  global = 1; /* dead store */
  global = 2;
  return;
  global = 3; /* unreachable */
}
```

will be changed to

```
int global;

void f () {
  global = 2;
  return;
}
```

8.1.3 Copy-Propagation

```
int f() {
  int i, j;
  i = 10;
  j = i;
  return j;
}
```

will be changed to

```
int f() {
  int i, j;
  i = 10;
  j = 10;
  return 10;
}
```

Note: the dead stores created by this copy propagation will be eliminated by dead-code elimination.

8.1.4 Loop Optimizations

Two types of loop optimizations are done by SDCC loop invariant lifting and strength reduction of loop induction variables. In addition to the strength reduction the optimizer marks the induction variables and the register allocator tries to keep the induction variables in registers for the duration of the loop. Because of this preference of the register allocator, loop induction optimization causes an increase in register pressure, which may cause unwanted spilling of other temporary variables into the stack / data space. The compiler will generate a warning message when it is forced to allocate extra space either on the stack or data space. If this extra space allocation is undesirable then induction optimization can be eliminated either for the entire source file (with --noinduction option) or for a given function only using #pragma noinduction.

Loop Invariant:

```
for (i = 0 ; i < 100 ; i ++)
    f += k + l;
```

changed to

```
itemp = k + l;
for (i = 0; i < 100; i++)
    f += itemp;
```

As mentioned previously some loop invariants are not as apparent, all static address computations are also moved out of the loop.

Strength Reduction, this optimization substitutes an expression by a cheaper expression:

```
for (i=0;i < 100; i++)
    ar[i*5] = i*3;
```

changed to

```
itemp1 = 0;
itemp2 = 0;
for (i=0;i< 100;i++) {
    ar[itemp1] = itemp2;
    itemp1 += 5;
    itemp2 += 3;
}
```

The more expensive multiplication is changed to a less expensive addition.

8.1.5 Loop Reversing

This optimization is done to reduce the overhead of checking loop boundaries for every iteration. Some simple loops can be reversed and implemented using a "decrement and jump if not zero" instruction. SDCC checks for the following criterion to determine if a loop is reversible (note: more sophisticated compilers use data-dependency analysis to make this determination, SDCC uses a more simple minded analysis).

- The 'for' loop is of the form

```
for(<symbol> = <expression>; <sym> [< | <=] <expression>; [<sym>++ | <sym> += 1])
<for body>
```

- The <for body> does not contain "continue" or 'break".
- All goto's are contained within the loop.
- No function calls within the loop.
- The loop control variable <sym> is not assigned any value within the loop
- The loop control variable does NOT participate in any arithmetic operation within the loop.
- There are NO switch statements in the loop.

8.1.6 Algebraic Simplifications

SDCC does numerous algebraic simplifications, the following is a small sub-set of these optimizations.

```
i = j + 0;      /* changed to:  */     i = j;
i /= 2;         /* changed to:  */     i >>= 1;
i = j - j;      /* changed to:  */     i = 0;
i = j / 1;      /* changed to:  */     i = j;
```

Note the subexpressions given above are generally introduced by macro expansions or as a result of copy/constant propagation.

8.1.7 'switch' Statements

SDCC changes switch statements to jump tables when the following conditions are true.

- The case labels are in numerical sequence, the labels need not be in order, and the starting number need not be one or zero.

```
switch(i) {
   case 4:  ...
   case 5:  ...
   case 3:  ...
   case 6:  ...
}
```

```
switch (i) {
    case 0:  ...
    case 1:  ...
    case 2:  ...
    case 3:  ...
}
```

Both the above switch statements will be implemented using a jump-table. The example to the right side is slightly more efficient as the check for the lower boundary of the jump-table is not needed.

- The number of case labels is at least three, since it takes two conditional statements to handle the boundary conditions.
- The number of case labels is less than 84, since each label takes 3 bytes and a jump-table can be utmost 256 bytes long.

Switch statements which have gaps in the numeric sequence or those that have more that 84 case labels can be split into more than one switch statement for efficient code generation, e.g.:

```
switch (i) {
  case 1:  ...
  case 2:  ...
  case 3:  ...
  case 4:  ...
  case 9:  ...
  case 10:  ...
  case 11:  ...
  case 12:  ...
}
```

If the above switch statement is broken down into two switch statements

```
switch (i) {
  case 1:  ...
  case 2:  ...
  case 3:  ...
  case 4:  ...
}
```

and

```
switch (i) {
  case 9:  ...
  case 10: ...
  case 11: ...
  case 12: ...
}
```

then both the switch statements will be implemented using jump-tables whereas the unmodified switch statement will not be. You might also consider inserting dummy cases 0 and 5 to 8 in this example.
The pragma nojtbound can be used to turn off checking the *j*ump *t*able *bound*aries. It has no effect if a default label is supplied. Use of this pragma is dangerous: if the switch argument is not matched by a case statement the processor will happily jump into Nirvana.

8.1.8 Bit-shifting Operations.

Bit shifting is one of the most frequently used operation in embedded programming. SDCC tries to implement bit-shift operations in the most efficient way possible, e.g.:

```
unsigned char i;
...
i >>= 4;
...
```

generates the following code:

```
mov  a,_i
swap a
anl  a,#0x0f
mov  _i,a
```

In general SDCC will never setup a loop if the shift count is known. Another example:

```
unsigned int i;
...
i >>= 9;
...
```

will generate:

```
mov  a,(_i + 1)
mov  (_i + 1),#0x00
clr  c
rrc  a
mov  _i,a
```

8.1.9 Bit-rotation

A special case of the bit-shift operation is bit rotation, SDCC recognizes the following expression to be a left bit-rotation:

```
unsigned char i;           /* unsigned is needed for rotation */
...
i = ((i << 1) | (i >> 7));
...
```

will generate the following code:

```
mov  a,_i
rl   a
mov  _i,a
```

SDCC uses pattern matching on the parse tree to determine this operation.Variations of this case will also be recognized as bit-rotation, i.e.:

```
i = ((i >> 7) | (i << 1)); /* left-bit rotation */
```

8.1.10 Nibble and Byte Swapping

Other special cases of the bit-shift operations are nibble or byte swapping, SDCC recognizes the following expressions:

```
unsigned char i;
unsigned int j;
...
i = ((i << 4) | (i >> 4));
j = ((j << 8) | (j >> 8));
```

and generates a swap instruction for the nibble swapping or move instructions for the byte swapping. The ”j” example can be used to convert from little to big-endian or vice versa. If you want to change the endianness of a *signed* integer you have to cast to `(unsigned int)` first.

Note that SDCC stores numbers in little-endian[1] format (i.e. lowest order first).

[1] Usually 8-bit processors don’t care much about endianness. This is not the case for the standard 8051 which only has an instruction to increment its *dptr*-datapointer so little-endian is the more efficient byte order.

8.1.11 Highest Order Bit

It is frequently required to obtain the highest order bit of an integral type (long, int, short or char types). SDCC recognizes the following expression to yield the highest order bit and generates optimized code for it, e.g.:

```
unsigned int gint;

foo () {
  unsigned char hob;
  ...
  hob = (gint >> 15) & 1;
  ..
}
```

will generate the following code:

```
                          61 ;  hob.c 7
000A E5*01                62          mov   a,(_gint + 1)
000C 23                   63          rl    a
000D 54 01                64          anl   a,#0x01
000F F5*02                65          mov   _foo_hob_1_1,a
```

Variations of this case however will *not* be recognized. It is a standard C expression, so I heartily recommend this be the only way to get the highest order bit, (it is portable). Of course it will be recognized even if it is embedded in other expressions, e.g.:

```
xyz = gint + ((gint >> 15) & 1);
```

will still be recognized.

8.1.12 Peephole Optimizer

The compiler uses a rule based, pattern matching and re-writing mechanism for peep-hole optimization. It is inspired by *copt* a peep-hole optimizer by Christopher W. Fraser (cwfraser@microsoft.com). A default set of rules are compiled into the compiler, additional rules may be added with the *--peep-file <filename>* option. The rule language is best illustrated with examples.

```
replace {
  mov %1,a
  mov a,%1
} by {
  mov %1,a
}
```

The above rule will change the following assembly sequence:

```
mov r1,a
mov a,r1
```

to

```
mov r1,a
```

Note: All occurrences of a *%n* (pattern variable) must denote the same string. With the above rule, the assembly sequence:

```
mov r1,a
mov a,r2
```

will remain unmodified.

Other special case optimizations may be added by the user (via *--peep-file option*). E.g. some variants of the 8051 MCU allow only `ajmp` and `acall`. The following two rules will change all `ljmp` and `lcall` to `ajmp` and `acall`

```
replace { lcall %1 } by { acall %1 }
replace { ljmp %1 } by { ajmp %1 }
```

The *inline-assembler code* is also passed through the peep hole optimizer, thus the peephole optimizer can also be used as an assembly level macro expander. The rules themselves are MCU dependent whereas the rule language infra-structure is MCU independent. Peephole optimization rules for other MCU can be easily programmed using the rule language.

The syntax for a rule is as follows:

```
rule := replace [ restart ] '{' <assembly sequence> '\n'
                            '}' by '{' '\n'
                                <assembly sequence> '\n'
                            '}' [if <functionName> ] '\n'
```

<assembly sequence> := assembly instruction (each instruction including labels must be on a separate line).

The optimizer will apply to the rules one by one from the top in the sequence of their appearance, it will terminate when all rules are exhausted. If the 'restart' option is specified, then the optimizer will start matching the rules again from the top, this option for a rule is expensive (performance), it is intended to be used in situations where a transformation will trigger the same rule again. An example of this (not a good one, it has side effects) is the following rule:

```
replace restart {
  pop %1
  push %1 } by {
  ; nop
}
```

Note that the replace pattern cannot be a blank, but can be a comment line. Without the 'restart' option only the innermost 'pop' 'push' pair would be eliminated, i.e.:

```
pop ar1
pop ar2
push ar2
push ar1
```

would result in:

```
pop ar1
; nop
push ar1
```

with the restart option the rule will be applied again to the resulting code and then all the pop-push pairs will be eliminated to yield:

```
; nop
; nop
```

A conditional function can be attached to a rule. Attaching rules are somewhat more involved, let me illustrate this with an example.

```
replace {
     ljmp %5
%2:
} by {
     sjmp %5
%2:
} if labelInRange
```

The optimizer does a look-up of a function name table defined in function *callFuncByName* in the source file SDCCpeeph.c, with the name *labelInRange*. If it finds a corresponding entry the function is called. Note there can be no parameters specified for these functions, in this case the use of *%5* is crucial, since the function *labelInRange* expects to find the label in that particular variable (the hash table containing the variable bindings is passed as a parameter). If you want to code more such functions, take a close look at the function labelInRange and the calling mechanism in source file SDCCpeeph.c. Currently implemented are *labelInRange, labelRefCount, labelIsReturnOnly, operandsNotSame, xramMovcOption, 24bitMode, portIsDS390, 24bitModeAndPortDS390* and *notVolatile*.

I know this whole thing is a little kludgey, but maybe some day we will have some better means. If you are looking at this file, you will see the default rules that are compiled into the compiler, you can add your own rules in the default set there if you get tired of specifying the --peep-file option.

8.2 ANSI-Compliance

Deviations from the compliance:

- functions are not always reentrant.
- structures cannot be assigned values directly, cannot be passed as function parameters or assigned to each other and cannot be a return value from a function, e.g.:

```
struct s { ...  };
struct s s1, s2;
foo()
{
    ...
    s1 = s2 ; /* is invalid in SDCC although allowed in ANSI */
    ...
}
struct s foo1 (struct s parms) /* invalid in SDCC although allowed in ANSI
   */
{
    struct s rets;
    ...
    return rets;/* is invalid in SDCC although allowed in ANSI */
}
```

- 'long long' (64 bit integers) not supported.
- 'double' precision floating point not supported.
- No support for setjmp and longjmp (for now).
- Old K&R style function declarations are NOT allowed.

```
foo(i,j) /* this old style of function declarations */
int i,j; /* are valid in ANSI but not valid in SDCC */
{
    ...
}
```

- functions declared as pointers must be dereferenced during the call.

```
int (*foo)();
...
/* has to be called like this */
(*foo)(); /* ANSI standard allows calls to be made like 'foo()' */
```

8.3 Cyclomatic Complexity

Cyclomatic complexity of a function is defined as the number of independent paths the program can take during execution of the function. This is an important number since it defines the number test cases you have to generate to validate the function. The accepted industry standard for complexity number is 10, if the cyclomatic complexity reported by SDCC exceeds 10 you should think about simplification of the function logic. Note that the complexity level is not related to the number of lines of code in a function. Large functions can have low complexity, and small functions can have large complexity levels.

SDCC uses the following formula to compute the complexity:

complexity = (number of edges in control flow graph) - (number of nodes in control flow graph) + 2;

Having said that the industry standard is 10, you should be aware that in some cases it be may unavoidable to have a complexity level of less than 10. For example if you have switch statement with more than 10 case labels, each case label adds one to the complexity level. The complexity level is by no means an absolute measure of the algorithmic complexity of the function, it does however provide a good starting point for which functions you might look at for further optimization.

8.4 Retargetting for other Processors

The issues for retargetting the compiler are far too numerous to be covered by this document. What follows is a brief description of each of the seven phases of the compiler and its MCU dependency.

- Parsing the source and building the annotated parse tree. This phase is largely MCU independent (except for the language extensions). Syntax & semantic checks are also done in this phase, along with some initial optimizations like back patching labels and the pattern matching optimizations like bit-rotation etc.
- The second phase involves generating an intermediate code which can be easy manipulated during the later phases. This phase is entirely MCU independent. The intermediate code generation assumes the target machine has unlimited number of registers, and designates them with the name iTemp. The compiler can be made to dump a human readable form of the code generated by using the --dumpraw option.
- This phase does the bulk of the standard optimizations and is also MCU independent. This phase can be broken down into several sub-phases:

  Break down intermediate code (iCode) into basic blocks.
  Do control flow & data flow analysis on the basic blocks.
  Do local common subexpression elimination, then global subexpression elimination
  Dead code elimination
  Loop optimizations
  If loop optimizations caused any changes then do 'global subexpression elimination' and 'dead code elimination' again.
- This phase determines the live-ranges; by live range I mean those iTemp variables defined by the compiler that still survive after all the optimizations. Live range analysis is essential for register allocation, since these computation determines which of these iTemps will be assigned to registers, and for how long.
- Phase five is register allocation. There are two parts to this process.

The first part I call 'register packing' (for lack of a better term). In this case several MCU specific expression folding is done to reduce register pressure.

The second part is more MCU independent and deals with allocating registers to the remaining live ranges. A lot of MCU specific code does creep into this phase because of the limited number of index registers available in the 8051.

- The Code generation phase is (unhappily), entirely MCU dependent and very little (if any at all) of this code can be reused for other MCU. However the scheme for allocating a homogenized assembler operand for each iCode operand may be reused.
- As mentioned in the optimization section the peep-hole optimizer is rule based system, which can reprogrammed for other MCUs.

Stand: August 2004, Version 8.0

© by

dr. rakers
MIKROCONTROLLER

Im Hemmen 2
59387 Ascheberg

e-mail: info@rakers.de
www: http://www.rakers.de

Das Handbuch unterliegt wie das Programm dem Urheberrecht. Kopieren ist nur zum eigenen Gebrauch erlaubt. Auszugsweises Vervielfältigen, unabhängig vom gewählten Verfahren, ist nur bei schriftlicher Erlaubnis des Autors gestattet.

Die Laufzeitbibliothek des BASIC und das Betriebsprogramm dürfen in Ihre Anwendungen eingebunden und zusammen mit diesen vertrieben werden. Bedingung ist, dass entweder Ihr Name (oder der Ihrer Firma) explizit in einem Copyright-Hinweis erscheint, oder dass der Name des CESY-Autors dort genannt wird. Die Laufzeitbibliothek darf nicht als eigenständiges Paket oder „Toolbox“ weitergegeben werden. Sowohl Laufzeitbibliothek (CLIB.SRC) als auch das Betriebsprogramm (SYS51.SRC, SYS535.SRC, SYS537,SRC, LAB537.SRC etc.) dürfen nur als Objektcode in Anwendungen eingebunden werden. Der mitgelieferte Assembler-Sourcecode hat bei Ihnen zu bleiben. Gleiches gilt auch für die Demonstrations-Programme. Über Ausnahmen ist mit dem Autor zu verhandeln.

Der SDCC-Compiler ist durch die GNU General Public License geschützt. Das SDCC-Paket darf vom Benutzer gratis benutzt werden. Etwaige Änderungen sind allen interessierten weiteren Nutzern zugänglich zu machen. Dies gilt jedoch nicht für mit dem SDCC/CESY-System übersetzte Programme. Diese verbleiben natürlich im Copyright des jeweiligen Autors.

Haftungsausschluss: Die Firma Rakers lehnt jegliche Ansprüche auf Schadenersatz aufgrund von Softwarefehlern ab. Genaueres siehe unsere allgemeinen Geschäftsbedingungen.